

### 版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF



总有人为你制定规则，总有人试图给你答案，但没有人可以代替你思考。

——韩寒

# 持续演进的 Cloud Native 云原生架构下微服务最佳实践

王启军 / 著



中国工信出版集团



電子工業出版社  
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY  
<http://www.phei.com.cn>



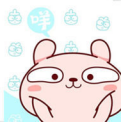


## 关于作者



### 王启军

目前就职于华为公司架构部，负责华为公司的Cloud Native、微服务架构推进落地，前后参与了华为手机祥云4.0、物联网IoT 2.0的架构设计。曾任当当架构师，主导电商平台架构设计，包括订单、支付、价格、库存、物流等。曾就职于搜狐，负责手机微博的研发。十余年的技术历练，也曾作为技术负责人带领过近百人的团队。公众号“奔跑中的蜗牛”的作者。







# 持续演进的 Cloud Native 云原生架构下微服务最佳实践

王启军 / 著

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING





## 内 容 简 介

本书从架构、研发流程、团队文化三个角度详细介绍了如何构建 Cloud Native。作者长期活跃在研发一线，具有丰富的架构设计经验，也曾亲身经历过很多失败的架构设计，如很多团队在实施微服务架构的时候，只强调拆分服务，根本没有理解微服务架构应该怎么做。本书就是想告诉读者，除了拆分服务，还要把哪些事做好，例如基础设施、一致性、性能、研发流程、团队文化等。

本书共分为 10 章，第 1 章从整体上描述了 Cloud Native 的起源、组成及原则等；从第 2 章到第 7 章重点描述了微服务架构、敏捷基础设施及公共基础服务、可用性、可扩展性、性能、一致性等方面的设计实践；第 8 章介绍了 Serverless 和 Service Mesh；第 9 章介绍了如何构建研发流程；第 10 章介绍了如何建设团队文化。

本书希望给技术管理者、架构师和有一定基础的技术人员提供帮助，特别是希望改变研发模式，从交付型软件过渡到云服务的传统软件企业开发者，此书将帮助你少走弯路。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

### 图书在版编目（CIP）数据

持续演进的 Cloud Native：云原生架构下微服务最佳实践 / 王启军著. —北京：电子工业出版社，2018.10  
ISBN 978-7-121-35120-4

I. ①持… II. ①王… III. ①程序语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字（2018）第 222022 号

责任编辑：汪达文

印 刷：三河市华成印务有限公司

装 订：三河市华成印务有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编：100036

开 本：787×980 1/16 印张：19.5 字数：410 千字

版 次：2018 年 10 月第 1 版

印 次：2018 年 10 月第 1 次印刷

印 数：2500 册 定价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 [zltts@phei.com.cn](mailto:zltts@phei.com.cn)，盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

本书咨询联系方式：（010）51260888-819，[faq@phei.com.cn](mailto:faq@phei.com.cn)。







# 推荐序一

## 云端应用时代之光

写书殊为不易，分享精神更是难能可贵。我一直坚定地认为，能把自己的技术经验写成一本公开出版的技术书籍是件非常了不起的事，因此对写书的朋友总会肃然起敬，对创作了多本技术书籍的朋友更是高山仰止。启军是我在当当时的同事，我对他的独到思维和行动力印象深刻，作为架构师他推动了产品线架构的演化升级，这样的同事令人兴奋。如今他从一线实践经验出发，结合行业主流技术发展趋势总结出自己的系统思考。具体而言，启军为云原生技术架构体系著书立说，在 IT 技术的浪潮之巅做布道师，给了我一个大大的惊喜。与这样努力的朋友为伍，我哪敢不持续学习和成长呢？

云原生架构是 IT 技术在云计算时代的进化升级，标志着云端应用进入成熟阶段。技术的价值是高效稳定、快速响应、驱动甚至引领业务发展，避免叠见层出，以及减少工作量。成规模的系统和团队需要与之匹配的技术体系。云计算兴起之时，有人说：“未来技术人员会分成两种，一种是构建云的，另一种是基于云构建应用的”。那时还没有成熟的云解决方案，对云计算的畅想也只能局限于原有的技术产品。如今云计算时代已经到来，应运而生并经过时间锤炼的云原生技术是这个时代的热点，因此技术人员只有与时俱进、更新技能，才能走向未来。

架构持续演进，技术也会更新换代，也许几年后书中的许多名词都会成为历史，而书





中所介绍的架构处理方法和思考问题的角度依然有借鉴意义。互联网时代的 IT 技术一直在加速演进，多种产品协同形成体系，技术人员进阶需要掌握整套技术栈。通过本书，我们不仅能够掌握云原生的各个关键组件，更能理解架构设计的思想，掌握技术架构持续演进的过程，以及其与团队文化、研发流程的共存互生的关系。从开发编码晋级到架构设计，需要提升认知、开阔视野，相信会有很多同道中人，在多年后回首来时的路，会想起这本书对自己的帮助，想起那些在字里行间有所得的刹那，心中暗喜，转身昂首阔步向前！

史海峰

公众号“IT 民工闲话”作者







## 推荐序二

有幸和启军共事一年多，我总是被他执着的工作态度和敏锐的技术洞察力深深震撼。启军在技术分享时说过一句话“有人告诉你答案，但没人代替你思考”，这句话引发了大家的共鸣，而他对业务的思考十分全面，推动了公司整体架构水平的持续提升。本书中提到的分布式 UUID、JOB 调度、消息队列和集群缓存，不仅依然在高效平稳地运行，而且帮助公司架构从基础组件演进到了云平台，并对外提供平台级服务。这充分展现了启军对架构持续演进的前瞻性。启军对业务端服务的剥离和抽象使得上百万数量级的商品变价可以实时实现，时效性和高可用在启军的设计中体现得淋漓尽致，这些在书中都有介绍。

启军耿直的工作作风和开放的心态，使得大家与他的合作都非常高效。各服务场景自己验证压力情况，破坏性测试恢复机制，如何做平衡方案，本书也都有详细介绍。这本书真正体现了实践的价值，为架构的持续演进做了铺路石，更为依然奋斗在云原生路上的朋友们提供了实践指南。不论是刚开始接触业务开发的同学，还是已经有多年实战经验的专家，阅读本书都会有醍醐灌顶之感。

刘利川

当当高级技术总监





# 序

## 架构没有绝对的对与错

在技术的领域里，并不存在“上帝”。没有人的每句话都是对的，没有人的所有思想都能被别人所接受。

我经常在公司范围内培训，首先是灌输架构思想和解决方案，然后会在实战演练中模拟一个比较简单的业务场景，把所有人分成 4 个团队，每个团队大概有 10 个人。结果发现，每个团队最终形成的架构图总有很大差异，很难评价一个团队的做法是对是错。例如，是要拆分为 3 个服务，还是 5 个服务，他们有各自的理由，除非比较明显的问题，否则你很难以一个理由去否定另一个理由。原因只是各个团队站在了不同的维度综合判断、权衡，形成了自己认为满意的架构方案。因此，架构没有绝对的对与错，只是在不同的角度做出的决定而已。

## 架构很难被衡量

每个公司的管理层都希望尽可能地去衡量架构的先进性，希望认清差距，向着好的架构方向不断演进。然而架构很难被衡量，须同时具备差距特别明显、制定指标的人能力达到一定高度、业务场景比较接近这三条才有可能衡量。当然我们可以去制定一些指标，这些指标应该是参考性的，作为一个自检项，而不是评价标准。从这个角度看，并不是符合 Cloud Native 就是好的，不符合就是差的，当不符合时，你的理由是什么？你站在问题的哪个角度？

Martin Fowler 曾说：“优秀的技术人员的观点胜过任何度量，尽管它是主观的。”

因为你无法统一每个人关注的点，以及对各自关注的点的重视程度，所以架构很难被衡量。







## 架构需要持续演进

在传统企业中，架构设计是一个很重要且很耗费时间的过程，需要经过很多轮审核，架构文档动辄几百页，而且这个文档绝对不能有没考虑到的问题，必须面面俱到、接近完美。例如，目前系统还没有用户，就要为未来 1 千万的用户耗费精力解决性能问题，而且软件永远有你想象不到的问题发生。实际上我们描述的是一种静止的架构，这种架构每次变更都需要耗费巨大的成本。如果此时恰好出现了一个基于敏捷思想的竞争对手，则会形成一种鲜明的对比，他们不去考虑太长时间之后的事，出现什么问题就解决什么问题，因为有可能一年以后这个项目死了，也有可能用户人数突破 1 亿，系统需要进行大规模重构。总之，未来是不确定的。可见，架构是锤炼出来的，而不仅是设计出来的。

反应速度是传统企业的硬伤，这不是通过加班就能解决的。可以看一下互联网巨头们每年的发布次数，动辄每年发布几百万乃至上千万次，每个服务每天都在发生变化，每周可能都会上线。在如今这个快速发展的世界里，你无法依赖一个人去做所有的决策。这就需要发挥所有成员的主观能动性，也就是说，架构应该交给一线决策。回到前面提到的问题，服务怎么拆分更好？我想只有深入了解需求、场景、目标甚至自身条件之后才能做出决策。并且，架构的演进不是一蹴而就的，而是一个长期发展的过程。

## 变革需要坚决

历史上的变革大多阻力重重，因为一旦变革就意味着打破原有的“默契”，打破原有的“潜规则”，而“顽固派”通常是原有文化的受益者，他们通常不会反对变革，而是通过“我们不能完全照抄，要走出适合我们的路”来促成妥协。如果变革过程中遇到任何风吹草动，就更会给“顽固派”各种理由“走自己的路”。这也就是为什么我们熟知世界领先 IT 企业的技术、研发流程和企业文化，而就是学不会的原因。

这时候需要的是企业领导者的果断、坚决。只要方向没错，就要坚持，决不动摇。下面这段话是马云对刘振飞（阿里技术保障部负责人）关于阿里云内部争议的回复，反映了一个领导者在企业变革过程中起到的作用。

在王坚加入阿里之前，我跟教授（指曾鸣）讨论公司的未来，觉得云计算和大数据代表未来，对国家和社会的发展有长远的意义，所以我们要干，这是第一点。但是怎么做云计算、大数据？我们谁也不知道。现在来了个人叫王坚，他说：“我知道怎么做”，为什么不支持呢？这是第二点。第三点，即使万一做失败了，那也没关系，咱们的人倒下 70%，还有 30%活着，咱们活下来的人打扫战场，换个方向继续干，总要把它做出来。





## 写代码不同于搬砖

如果是搬砖，那么效率高的人和效率低的人之间的差距不会太大，因此每个人每天的工资都是相对固定的。但是在如今这个知识爆炸的时代，对于从事软件行业的群体来说，效率高者的工作效率比效率低者的可能高出几十倍、几百倍，优秀的人能写出更高质量的代码，能够预测问题。而在这个行业越是优秀的人才越是稀缺，因此很多互联网公司都愿意花大价钱去招一些更优秀的人。

优秀的人不愿意来，不一定是因为钱。花钱雇佣优秀的人是一方面，怎样管理这些人又是另外一方面，用管理搬砖者的方式来管理他们是不行的，管理优秀的人需要给予他们更多的信任，需要营造一种公开透明、自由高效的环境。

## 关于本书

为什么会出现 Cloud Native 这个概念呢？无论是云化、平台化，还是微服务架构，又或者是敏捷开发、自动化，都只是描述了几个点，而 Cloud Native 更像是一个面，通过它把这些点都关联起来了。某几个点做得很好而忽略了其他点通常会走入误区。例如，某些团队只关注服务拆分，而忽略了工具、组织对微服务的影响，最终效果并不理想。又如，要提升系统的可用性，只是从技术的角度去考虑是不够的，还要考虑如何通过自动化测试提升可用性，如何通过 Code Review 提升可用性，以及当故障发生时如何快速修复。我希望通过个人的工作经历以书的方式传递一些这方面的经验教训。

本书分别从架构、研发流程、团队文化三个角度全面论述 Cloud Native，因为只有三方面配合才能达到理想的效果。我见到过无数失败的案例，绝大多数都是因为考虑得比较片面，例如单纯从架构角度进行变革，或者单纯从研发流程角度变革。我们希望模仿 Google、Facebook、Amazon、Netflix 等领先企业，但是往往高估了架构的影响力，而低估了研发流程和团队文化的影响力。实际上，研发流程和团队文化对架构有着非常重要的影响。本书以 Cloud Native 的起源、诉求及组成开始，全面描述了 Cloud Native 的各个方面。从架构角度阐述了如何实施微服务架构，如何构建敏捷基础设施及平台服务。同时，从可用性、可扩展性、性能、一致性等角度描述了微服务架构中产生的问题及解决方案。最后，分别描述了 Cloud Native 下的研发流程和团队文化。

本书比较适合技术管理者、架构师和有一定基础的技术人员阅读，特别是传统软件企业的技术领导者，以及希望向互联网公司转型的或者转型失败的企业技术领导者。此书将帮助这些人少走弯路。还有一些比较有经验的高级研发人员，阅读此书也利于系统掌握 Cloud Native 的关键技能。无论如何，都希望此书能给你带来较好的体验，使你获得启发。

书中的内容大多来自我的工作经验，不免存在遗漏及错误，欢迎指正。读者可以直接





发送邮件到我的邮箱 (41309975@qq.com)，在此提前表示感谢。

感谢工作中的各位同事、生活中的各位好友，正是他们的支持和鼓励，才让本书完成。更要感谢我的家人，特别是我的妻子和女儿，是她们的拥抱和灿烂的笑容让我坚定了信念。

王启军

轻松注册成为博文视点社区用户 (www.broadview.com.cn)，扫码直达本书页面。

- **提交勘误：**您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动：**在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/35120>





# 目 录

第 1 章 综述	1
1.1 Cloud Native 的起源	1
1.2 Cloud Native 的组成	4
1.3 Cloud Native 背后的诉求	5
1.4 如何衡量 Cloud Native 的能力	5
1.5 Cloud Native 的原则	6
第 2 章 微服务架构	11
2.1 微服务架构的起源	11
2.2 为什么采用微服务架构	12
2.2.1 单体架构与微服务架构	12
2.2.2 什么时候开始微服务架构	14
2.2.3 如何决定微服务架构的拆分粒度	14
2.3 微服务设计原则	15
2.4 微服务架构实施的先决条件	17
2.4.1 研发环境和流程上的转变	17
2.4.2 拆分前先做好解耦	18
2.5 微服务划分模式	20
2.5.1 基于业务复杂度选择服务划分方法	20
2.5.2 基于数据驱动划分服务	21
2.5.3 基于领域驱动划分服务	22
2.5.4 从已有单体架构中逐步划分服务	23
2.5.5 微服务拆分策略	24
2.5.6 如何衡量服务划分的合理性	25
2.6 微服务划分反模式	26

2.7 微服务 API 设计	28
2.7.1 优秀 API 的设计原则	28
2.7.2 服务间通信——RPC	28
2.7.3 序列化——Protobuf	30
2.7.4 服务间通信——RESTful	33
2.7.5 通过 Swagger 实现 RESTful	36
2.7.6 通过 Spring Boot、Springfox、Swagger 实现 RESTful	41
2.7.7 HTTP 协议的进化——HTTP/2	46
2.7.8 HTTP/2 和 Protobuf 的组合——gRPC	48
2.8 微服务框架	53
2.9 基于 Dubbo 框架实现微服务	54
2.10 基于 Spring Cloud 框架实现微服务	58
2.11 服务发现场景下的 ZooKeeper 与 Etcd	67
2.12 微服务部署策略	68
2.12.1 服务独享数据库	69
2.12.2 服务独享虚拟机/容器	70
2.13 为什么总觉得微服务架构很别扭	70
第 3 章 敏捷基础设施及公共基础服务	73
3.1 传统基础设施面临的挑战	73
3.2 什么是敏捷基础设施	74
3.3 基于容器的敏捷基础设施	75
3.3.1 容器 VS 虚拟机	76
3.3.2 安装 Docker	77
3.3.3 部署私有 Docker Registry	79
3.3.4 基于 Spring Boot、Maven、Docker 构建微服务	79
3.3.5 基于 docker-compose 管理容器	84
3.4 基于公共基础服务的平台化	85
3.5 监控告警服务	86
3.5.1 监控数据采集	87
3.5.2 监控数据接收模式	87
3.5.3 通过时间序列数据库存储监控数据	88
3.5.4 开源监控系统实现 Prometheus	88
3.5.5 通过 Prometheus 和 Grafana 监控服务	90
3.6 分布式消息中间件服务	96
3.6.1 分布式消息中间件的作用	97

3.6.2	业界常用的分布式消息中间件	98
3.6.3	Kafka 的设计原理	99
3.6.4	为什么 Kafka 性能高	100
3.6.5	Kafka 的数据存储结构	102
3.6.6	如何保证 Kafka 不丢消息	104
3.6.7	Kafka 跨数据中心场景集群部署模式	106
3.7	分布式缓存服务	108
3.7.1	分布式缓存的应用场景	109
3.7.2	业界常用的分布式缓存 Memcached	110
3.7.3	业界常用的分布式缓存——Redis	111
3.7.4	Redis 常用的分布式缓存集群模式	112
3.7.5	基于 Codis 实现 Redis 分布式缓存集群	116
3.8	分布式任务调度服务	118
3.8.1	通过 Tbschedule 实现分布式任务调度	119
3.8.2	通过 Elastic-Job 实现分布式任务调度	123
3.9	如何生成分布式 ID	126
3.9.1	UUID	126
3.9.2	SnowFlake	127
3.9.3	Ticket Server	128
3.9.4	小结	129
第 4 章	可用性设计	130
4.1	综述	130
4.1.1	可用性和可靠性的关系	130
4.1.2	可用性的衡量标准	131
4.1.3	什么降低了可用性	131
4.2	逐步切换	132
4.2.1	影子测试	132
4.2.2	蓝绿部署	133
4.2.3	灰度发布/金丝雀发布	134
4.3	容错设计	135
4.3.1	消除单点	136
4.3.2	特性开关	136
4.3.3	服务分级	137
4.3.4	降级设计	138
4.3.5	超时重试	139

4.3.6 隔离策略	152
4.3.7 熔断器	153
4.4 流控设计	157
4.4.1 限流算法	157
4.4.2 流控策略	159
4.4.3 基于 Guava 限流	160
4.4.4 基于 Nginx 限流	162
4.5 容量预估	163
4.6 故障演练	164
4.7 数据迁移	165
4.7.1 逻辑分离, 物理不分离	166
4.7.2 物理分离	166
第 5 章 可扩展性设计	168
5.1 加机器能解决问题吗	168
5.2 横向扩展	169
5.3 AKF 扩展立方体	170
5.4 如何扩展长连接	172
5.5 如何扩展数据库	175
5.5.1 X 轴扩展——主从复制集群	175
5.5.2 Y 轴扩展——分库、垂直分表	176
5.5.3 Z 轴扩展——分片 (sharding)	177
5.5.4 为什么要带拆分键	182
5.5.5 分片后的关联查询问题	183
5.5.6 分片扩容 (re-sharding)	184
5.5.7 精选案例	187
5.6 如何扩展数据中心	190
5.6.1 两地三中心和同城多活	190
5.6.2 同城多活	191
5.6.3 异地多活	192
第 6 章 性能设计	194
6.1 性能指标	195
6.2 如何树立目标	195
6.3 如何寻找平衡点	196
6.4 如何定位瓶颈点	197
6.5 服务通信优化	198



6.5.1	同步转异步	198
6.5.2	阻塞转非阻塞	199
6.5.3	序列化	200
6.6	通过消息中间件提升写性能	201
6.7	通过缓存提升读性能	202
6.7.1	基于 ConcurrentHashMap 实现本地缓存	203
6.7.2	基于 Guava Cache 实现本地缓存	204
6.7.3	缓存的常用模式	205
6.7.4	应用缓存的常见问题	207
6.8	数据库优化	208
6.8.1	通过执行计划分析瓶颈点	208
6.8.2	为搜索字段创建索引	209
6.8.3	通过慢查询日志分析瓶颈点	210
6.8.4	通过提升硬件能力优化数据库	211
6.9	简化设计	212
6.9.1	转移复杂度	212
6.9.2	从业务角度优化	212
第 7 章	一致性设计	214
7.1	问题起源	214
7.2	基础理论	215
7.2.1	什么是分布式事务	216
7.2.2	CAP 定理	218
7.2.3	BASE 理论	219
7.2.4	Quorum 机制（NWR 模型）	219
7.2.5	租约机制（Lease）	220
7.2.6	状态机（Replicated State Machine）	221
7.3	分布式系统的一致性分类	222
7.3.1	以数据为中心的一致性模型	223
7.3.2	以用户为中心的一致性模型	226
7.3.3	业界常用的一致性模型	229
7.4	如何实现强一致性	230
7.4.1	两阶段提交	230
7.4.2	三阶段提交（3PC）	231
7.5	如何实现最终一致性	232
7.5.1	重试机制	232

7.5.2	本地记录日志	233
7.5.3	可靠事件模式	233
7.5.4	Saga 事务模型	235
7.5.5	TCC 事务模型	237
7.6	分布式锁	238
7.6.1	基于数据库实现悲观锁和乐观锁	239
7.6.2	基于 ZooKeeper 的分布式锁	241
7.6.3	基于 Redis 实现分布式锁	242
7.7	如何保证幂等性	244
7.7.1	幂等令牌 (Idempotency Key)	244
7.7.2	在数据库中实现幂等性	246
第 8 章	未来值得关注的方向	247
8.1	Serverless	247
8.1.1	什么是 Serverless	247
8.1.2	Serverless 的现状	248
8.1.3	Serverless 的应用场景	249
8.2	Service Mesh	250
8.2.1	什么是 Service Mesh	250
8.2.2	为什么需要 Service Mesh	252
8.2.3	Service Mesh 的现状	253
8.2.4	Istio 架构分析	255
第 9 章	研发流程	258
9.1	十二因子	258
9.2	为什么选择 DevOps	261
9.3	自动化测试	263
9.3.1	单元测试	263
9.3.2	TDD	264
9.3.3	提交即意味着可测试	265
9.4	Code Review	265
9.4.1	Code Review 的意义	265
9.4.2	Code Review 的原则	266
9.4.3	Code Review 的过程	267
9.5	流水线	267
9.5.1	持续交付	267
9.5.2	持续部署流水线	268

9.5.3 基于开源打造流水线	268
9.5.4 Amazon 的流水线	271
9.5.5 开发人员自服务	271
9.6 为什么需要 AIOps	272
9.7 基于数据和反馈持续改进	273
9.8 拥抱变化	274
9.9 代码即设计	274
第 10 章 团队文化	276
10.1 为什么团队文化如此重要	276
10.2 组织结构	278
10.2.1 团队规模导致的问题	278
10.2.2 康威定律	278
10.2.3 扁平化的组织	279
10.2.4 独裁的管理方式还是民主的管理方式	280
10.2.5 民主的团队如何做决策	282
10.3 环境氛围	282
10.3.1 公开透明的工作环境	282
10.3.2 学习型组织	283
10.3.3 减少正式的汇报	284
10.3.4 高效的会议	284
10.3.5 量化指标致死	286
10.4 管理风格	287
10.4.1 下属请假你会拒绝吗	287
10.4.2 为什么你招不到你想要的人	288
10.4.3 得到了所有人的认可，说明你并不是一个好的管理者	291
10.4.4 尽量避免用自己的权力去做决策	291
10.4.5 一屋不扫也可助你“荡平天下”	292
10.4.6 如何留下你想要的人	293
10.5 经典案例	294
10.5.1 Instagram 的团队文化	294
10.5.2 Netflix 的团队文化	294

# 1

## 第 1 章 综述

在介绍 Cloud Native 如何实现持续演进之前，先通过本章了解一下 Cloud Native，统一思想和术语，为后面章节的阅读打下基础。首先描述 Cloud Native 是怎么来的，承载着哪些诉求。然后简要介绍一下 Cloud Native 的组成，如何衡量其能力水平，以及 Cloud Native 下的相关原则。

### 1.1 Cloud Native 的起源

了解 Cloud Native 的起源能够帮助我们理解其含义。Cloud Native 和其他的架构概念不同，它的提出经历了很长的过程，每个人对于 Cloud Native 的理解也不尽相同。下面我们介绍其中比较著名的三位技术领导者关于 Cloud Native 的描述。

#### Paul Fremantle 提出的 Cloud Native

2010 年 5 月 28 日，WSO2 的 CTO 和联合创始人 Paul Fremantle 在他写的一篇博客中首次提出了 Cloud Native 这个概念。Paul Fremantle 提出 Cloud Native 的原因是他一直想用一个词表达一种架构，这种架构能描述应用程序和中间件在云环境中的良好运行状态。因此他抽象出了 Cloud Native 必须包含的属性，只有满足了这些属性才能保证良好的运行状态。



Paul Fremantle 对 Cloud Native 的属性总结如下。

- 分布式。
- 弹性。
- 多租户。
- 自服务。
- 按需计量和计费。
- 增量部署和测试。

## Adrian Cockcroft 提出的 Cloud Native

直到 2013 年，Netflix 的云架构师（2016 年 10 月他成为 AWS 的 VP）Adrian Cockcroft 在 Yow Conference 上介绍了 Netflix 在 AWS 上基于 Cloud Native 的成功应用。Netflix 在 AWS 上运行着上万个实例，每天都有数以千计的实例被创建或删除。Netflix 的成功，吸引着大批研发人员争相模仿，Adrian Cockcroft 介绍 Netflix 的成功经验时，主要从目标、原则和措施等方面进行了描述。

Adrian Cockcroft 对 Cloud Native 的目标总结如下。

- 可扩展性。对于 Netflix 这样的公司来说，可扩展性至关重要。据 Adrian Cockcroft 介绍，Netflix 在 AWS 上运行着几千个 Cassandra 节点和上万个服务实例。
- 高可用性。高可用代表了更好的用户体验，系统每分钟的不可用都意味着金钱上的损失。
- 敏捷。在互联网公司，速度永远是第一位的，速度代表了良好的用户体验。Netflix 利用了 AWS，而不是自建云环境及平台服务，这可以加快研发速度。
- 效率。在软件研发过程中，效率高的人比效率低的人效率不只是高出百分之二十，有可能是几十倍，甚至是上百倍。

为了达成以上目标，Netflix 制定了五大架构原则，通过这些架构原则约束所有研发人员的思想。Adrian Cockcroft 对 Netflix 实施 Cloud Native 的架构原则总结如下。

- 不变性。服务的实例一旦创建，将不能修改，如果要修改，则可以通过创建一个新的节点实现。
- 关注点分离。通过微服务架构实现关注点分离，避免出现“决策瓶颈”。实际上，实现关注点分离有助于提升系统的扩展性和可用性。
- 反脆弱性。默认所有的依赖都可能失效，在设计阶段就要考虑到如何处理这些失效问题。为了让系统更强壮，Netflix 会不断地攻击自己、主动破坏，以提醒系统要进行反脆弱性设计。

- 高信任的组织。Netflix 是基于信任的管理风格，相信自己的员工可以做出正确的决策，倡导给基层员工自主决策权。
- 共享。在 Netflix，管理是比较透明的，共享能够促进技术人员的成长。

那么 Netflix 通过什么措施实现了上面的目标呢？据 Adrian Cockcroft 介绍，Netflix 主要采取了如下措施。

- 利用 AWS 实现可扩展性、敏捷和共享。
- 利用非标准化数据实现关注点分离。
- 利用猴子工程师<sup>①</sup>实现反脆弱性。
- 利用默认开源实现敏捷、共享。
- 利用持续部署实现敏捷、不变性。
- 利用 DevOps 实现高信任组织和共享。
- 利用运行自己写的代码实现反脆弱性开发演进。

## Matt Stine 提出的 Cloud Native

最后一位比较有影响力的技术领导者是来自 Pivotal 的 Matt Stine，他在电子书 *Migrating to Cloud Native Application Architectures* 中对于如何将应用迁移到 Cloud Native 做了详细的介绍。Matt Stine 认为在单体架构向 Cloud Native 迁移的过程中，需要文化、组织、技术共同变革。该书把 Cloud Native 描述为一组最佳实践，包含如下几个重要内容。

- 十二因子。
- 微服务。
- 自服务敏捷基础设施。
- 基于 API 的协作。
- 反脆弱性。

还有一些组织和个人也给出了类似的定义和看法，但追求 Cloud Native 准确的定义并不是我们的初衷。业务场景是千变万化的，一方面，每个人站的角度不一样，心目中的架构也不同；另一方面，架构是在持续演进的，随着新的技术层出不穷，Cloud Native 留给世人的形象也会不断演进。

如果非要给 Cloud Native 下一个定义，那么我认为，Cloud Native 是一系列架构、研发流程、团队文化的最佳实践集合，以此支撑更快的创新速度、极致的用户体验、稳定可靠的用户服务、高效的研发效率。

---

① 英文为 Chaos Monkey，国内大多翻译为猴子工程师，意指通过有规则的破坏来发现系统脆弱性的工程师。

## 1.2 Cloud Native 的组成

观察任何一个企业都可以从三个角度出发，这三个角度分别是技术、流程、文化，三个方面都做好才能成为伟大的企业。Cloud Native 也一样，需要从架构、研发流程、团队文化三个角度来实现，三者需要相互配合，缺一不可。Cloud Native 的组成，如图 1-1 所示。

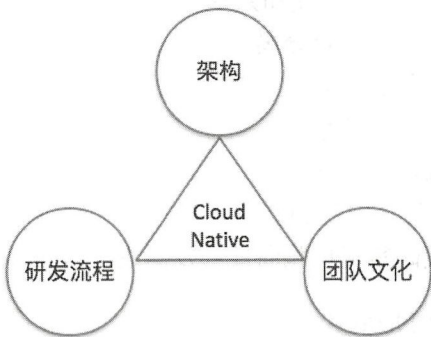


图 1-1 Cloud Native 的组成

从架构的角度来讲，Cloud Native 是以云和微服务架构为基础构建系统的，这里的云并不一定是公有云，也可以是私有云、混合云，云包含了敏捷基础设施及公共基础服务<sup>①</sup>。除此之外，还需要考虑架构的质量属性，本节将从一致性、性能、可扩展性、可用性等方面进行详细阐述。Cloud Native 架构的组成，如图 1-2 所示。

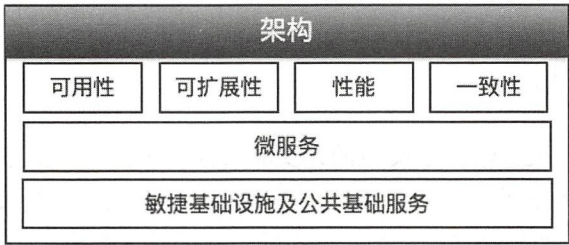


图 1-2 Cloud Native 架构的组成

从研发流程的角度来讲，自动化的研发环境是 Cloud Native 的基础。因为使用云作为基础设施，已经具备基础的自动化能力，可以达到自服务的要求，流程中应该尽量减少沟通人员的规模，尽量减少测试及运维对开发的协助，最好由全栈工程师独自、快速完成交

<sup>①</sup> 第 3 章详细介绍了敏捷基础设施及公共基础服务。

付。保持各个环境一致，容器使得这一点更容易实现。

从组织文化的角度来讲，一切以人为主，需要建立自由开放的环境，高度信任，增强自我驱动，充分发挥每一个人的力量，而不是让工程师变成“螺丝钉”。微服务架构要求小团队具有自主决策的权利，避免无论大事小事都要拿到会议上讨论，造成决策瓶颈。

## 1.3 Cloud Native 背后的诉求

在 Amazon 曾经出现过一个这样的问题，当团队人数快速增长之后，沟通效率越来越低，如何提高沟通效率呢？如何减少会议呢？最好的方式当然是不开会，那么怎么才能不开会却实现高效沟通呢？那就是让各个团队关注不同的模块。拆分服务就是一种方式，让每个团队独立负责一个服务，通过契约化的接口缩小沟通范围，只要接口不发生变化，就不需要过分关注外部的变化。这就是 Amazon 拆分服务的初衷。

现在正处于一个业务快速增长的时代，产品需要更快的交付速度，更好的用户体验，机会转瞬即逝。为什么传统企业打不过互联网公司呢？一个重要的原因是产品的进化速度太慢，不能根据用户的反馈快速迭代，当某个功能用户使用的频率比较高时，产品的方向可能会发生转变。

交付速度的提高不能以降低可用性为代价。我们都知道更新是可用性的“天敌”，只要更新就可能发生故障。传统企业提升可用性的一种方法就是少发布、多审核，这显然是和高交付速度背道而驰的。Cloud Native 通过一系列工具、方法减少发布导致的可用性问题，而不是减少发布次数。

在微服务架构中，服务数量大幅增加，性能、一致性等问题越来越严重，架构变得越来越复杂，如何解决这些问题？在本书中可以找到想要的答案。

## 1.4 如何衡量 Cloud Native 的能力

实际上，以成熟度模型来描述 Cloud Native 要优于使用精确的定义。表 1-1 通过 Cloud Native 采用的技术点来描述架构的等级。当然有可能你目前处于第一级，但是已经使用了一些第三级的技术点，这没有关系，这个表只是起到一个参考作用，并不要求严格按照其中的等级执行。也就是说，哪怕现在处于第一级，也完全可以直接跳过第二级到第三级。但是一个系统的架构，通常有一个发展过程，并非一步到位的。如果采用比较成熟的公有云，实际上已经具备了第三级的能力，需要做的仅仅是在应用层面做到服务化。



表 1-1 Cloud Native 成熟度模型

成熟度	描 述	关键技术点	效 果
第四级	系统具有自学习、自诊断、自调整、自恢复能力； 高度可视化、自动化； 实现自动发布（AI 选择发布时间）、自动降级、自动回滚； 可根据 AI 自动调整参数，如超时时间； 所有公共服务形成统一的整体，通过接口实现数据共享，实时调整。例如监控可以反压流控	人工智能服务进行决策； 强大的公共基础服务； 智能化运维； 高度自动化能力； 可实现 Serverless 架构	AIOps/NoOps； 资源利用率极大提升； 瞬间扩展能力； 强大的可用性； 强一致性
第三级	可编程基础设施； 中间件作为后端服务 <sup>①</sup> 提供； 任何时刻业务无中断； 实现 1%-10%-100% 的灰度发布流程； 自动扩/缩容方式，包括有状态的数据库（自动化负载均衡、分库分表）、缓存； 自动化降级； 开发、测试、生产环境统一； 全球化的业务发布能力，异地多活	分布式数据库、 分布式存储、 分布式缓存、 分布式消息队列、 灰度发布平台、 全局容错方案、 全局一致性方案、 混沌测试、 容器、 资源调度平台	不需要择时发布； 开发人员专注于业务，架构能力由基础设施承载； 公共基础服务共享重用
第二级	微服务架构，服务满足无状态、自治、隔离的条件； 服务根据业务划分等级； 非核心业务可以实现快速降级； 不受失败服务的影响，快速隔离	微服务框架、 持续交付流水线、 调用链分析、 分布式配置、 自动化测试、 监控系统	交付周期提升明显； 小团队开发，沟通效率提升； 对测试人员依赖降低
第一级	单体架构或者大粒度的拆分； 应用运行在虚拟化的环境中； 应用可以通过镜像或脚本自动化部署	负载均衡服务、 模块化、 虚拟化隔离	瀑布式开发； 系统重构基本是革命式的； 系统可用性严重依赖运维人员

## 1.5 Cloud Native 的原则

在软件设计的过程中，团队会用一些原则来约束开发者，将其作为开发者共同遵守的

① 作为服务提供使用，开发、测试、部署、维护都有专门的团队负责，只需要通过接口调用，轻松实现多租户需求。

标准或规则。本节许多地方都会涉及原则，对原则的描述是为了阐释 Cloud Native 的宏观思想。

## 为失败设计原则

试想，如果让你设计一条非常可靠的海底铁路隧道，那么你应该怎么设计呢？因为这涉及很多人的生命安全，你必须消除所有失败的可能性。事实上，通常会建立三条相互连接的隧道，其中两条可以容纳铁路运输，另一条既可以用来维护，也可以在紧急情况下用于逃生。30 年前的英法海底隧道正是采用了这个设计思路。从架构的角度讲，为失败设计同样重要，因为失败是不可避免的，我们希望失败的结果是我们预料到的，是经过设计的。

在微服务架构场景中，当服务数量越来越多，依赖越来越复杂时，出现问题的概率会越来越大，问题定位也会越来越困难，这时候再用传统的解决方法将是一个灾难，传统方法通常将可靠性等同于防止故障，需要在思想上进行转变。微服务架构由于存在更多的远程调用，任何的外部依赖都有可能会失效或延迟，这是潜在的故障和瓶颈。例如，试图一味地提升硬盘的质量不如存多份数据更可靠，因为失败是不可避免的，所以设计目标是预测并解决这些故障。因此，设计服务时应充分考虑异常情况，从使用者的角度出发，能够容忍故障的发生，最小化故障的影响范围。例如，在电商系统中，价格很重要，如果价格服务整体不可用，那么这时候前端详情页显示一个没有价格的页面比一个什么也不显示或者显示后端服务失效的页面强得多。

## 不变性原则

我们已经听到过很多由于运维人员删错了文件，或者配置错了参数而导致的故障。对于资源调度来说，我们更希望所有的服务无差异化配置，所有的服务都是标准的，而不希望在部署任何服务的过程中还需要手动操作，手动操作是不容易回溯的。

实现不变性原则的前提是，基础设施中的每个服务、组件都可以自动安装、部署，不需要人工干预。每个服务或组件在安装、部署完成后将不会发生更改，如果要更改，则丢弃老的服务或组件并部署一个新的服务或组件。另外，为了提升可用性，我们应该尽量减少故障修复时间，要知道替换的速度远远快于修复的速度，这种思想与不可变对象<sup>①</sup>的概念完全相同。

---

① 在面向对象和函数式编程中，不可变对象（Immutable Object）是指一个对象在创建后，其状态无法修改。不可变对象更容易构造、测试及使用。

## 去中心化原则

中心化往往代表的是瓶颈点，在微服务场景下，每个服务可以独立采用自己的技术方案或技术栈，因为每个服务具有自己独立的业务场景，可以根据实际情况进行选择。服务之间通过进程隔离，每个服务都有独立的数据库，一个服务实例失效不会导致大规模故障。相对于单体架构，这是一种去中心化的设计，系统没有一个物理或者逻辑的中心控制节点，不会因为一个节点的故障导致整个系统不可用。

另外，从研发流程的角度来说，去中心化意味着关注点分离。在微服务场景下，每个服务由独立的全功能团队负责，相对来说，更容易实现关注点分离。团队具有决策权，每个服务可以独立的开发、测试、部署、升级，只要接口不发生变化，对其不必过度关心。

Cloud Native 对开发团队的一个非常重要的要求是独立自主。可以尝试反问团队：“服务是否经常必须经过团队外人员审批才能上线？”当然，责任和权力是相辅相成的，例如，开发团队需要回答这样的问题：系统由 Java 换成了 Go，是否具备充足的理由？一旦出现问题是否能解决？

## 标准化原则

很多人都听说过宠物和牲畜的故事。宠物是有名字的，宠物和人有感情，不能被随意替代，而牲畜只是产奶、产肉，很容易被替代，标准化能让软件更像牲畜，而不是宠物。当所有程序都非常标准的时候，采用自动化的手段管理更容易。例如，如果我们都采用相同的微服务框架，那么服务之间的调用将变得非常容易。而且，团队间发生人员流动，也不再会因为换了一种框架而需要漫长的熟悉时间。当所有的日志打印都遵循某种标准的时候，对于排除故障，日志分析将非常重要。

这些标准最好不是通过规范性的文档来实现。例如，公司的编程规范，大多数人只有入职时可能会看一遍。最好的方式是通过框架来固化这些标准，通过工具来检测这些标准。例如可以利用 SonarQube 检测代码重复度，用 CheckStyle 检测代码风格。

标准化包含的范畴非常广，从最简单的操作系统、HostName 到软件安装的目录、参数、版本、配置文件等，业界很少有统一的标准，因为很多系统存在历史遗留问题，标准化的成本比较高。但是，一旦系统规模变大，对可用性要求变高，做到标准化是无法逃避的。

独立自主和标准化是一对互斥的原则，独立代表的是灵活、创新，而标准则代表效率、稳定，两者需要权衡。所谓独立自主是在一定的标准下实现的。例如 JMS 规范，如果都遵循规范，就不会有目前炙手可热的 Kafka，又如 SQL92 与 MVCC (Multi-Version Concurrency Control, 多版本并发控制) 的关系，标准化不能成为一种束缚。

当公司内某个工具或者服务不成熟的时候，就会出现标准化的问题。随着工具或者服



务的成熟，标准化成为一种常态。例如公司内的微服务框架，最初通常是野蛮生长的，会出现很多版本，因为初期研发人员对微服务框架的理念理解不一。随着时间的推移，理念逐步统一，认识到一套框架就可以解决问题，既可以节省研发投入，又可以形成标准化。

## 速度优先原则

美国利宝相互保险公司执行副总裁兼首席信息官 James McGlennon 说过：“如果你不能提高上市速度，毫无疑问，市场将发生变化，无论你对产品的设计、构建、部署或对员工的培训有多好，都不会完全适合市场需求，只因为晚了一点点。”

如果要赢得客户，赢得市场，速度无疑是更重要的，因为你要比竞争对手更快。速度和效率并不总是矛盾的，效率在大多数场景下是为速度提供条件的，效率有时意味着更多的流程，更多的依赖，效率更像一种“节流”方法，而速度是接近于“开源”的一种手段。当速度和效率发生冲突时，速度优先。

这就是你经常见到的，在微服务架构的场景下，有时会存在重复代码、重复开发，这样做的原因是为了减少依赖，使系统可以更独立。

## 简化设计原则

NBA 的数据统计非常全面，评价一个控卫的非常重要的指标之一就是助攻失误比。相应的，一个球队的助攻失误比也经常被作为球队的衡量指标。当追求更多助攻的时候，就会产生更多失误，因此当进行到关键阶段，如 5 分钟内差距小于 5 分的时候，球队一般都会选择单打，减少传球，目的是为了减少失误。架构也是一样，越是基础的服务，越需要稳定，越需要简化设计、简化运维。简化设计也是 Amazon 和 Netflix 的软件设计原则。

如果你发现设计满足了所有的要求，说明你一定过度设计了。你可能花了更多的时间，设计了一个非常复杂的系统，这对无论是开发复杂度还是工作量，都是一个巨大的挑战。所有的架构都应该本着简单的原则，在架构设计之后，应该多次简化架构设计方案。当然这并不表示所有的地方都追求简单，而是要有所侧重。根据帕累托法则（又称二八定律），80%的产出源自 20%的投入。我们首先要简化范围，找到那 20%的需要努力的点。通过这种方式找到核心业务流程，投入 80%的时间去设计它。

一个非常重要的检测方法是，产品经理的所有想法都实现了吗？按照以上逻辑，大多数产品经理提出的不合理的业务实现应该在发布到线上之前被否定，因为一旦上线，再下线就非常麻烦，会损害一部分用户的体验。



## 自动化驱动原则

波音公司分析了从 1995 年到 2005 年的数据，发现 55% 的生产事故是人为原因导致的，而根据《SRE Google 运维解密》一书中的描述，70% 生产事故来源于部署变更。传统行业做到高可靠更多的是通过严格的流程把控，严控每个环节的质量，减少变更次数。为了减少故障，投入更多的测试人员，甚至通过绩效“威胁”研发人员提升质量，这势必带来速度上的劣势，这就是为什么传统行业的公司见到互联网公司感觉用不上力的原因，这是长期积累形成的一种文化，公司的“DNA”很难改变。

部署与运维的成本会随着服务的增多呈指数级增长，每个服务都需要部署、监控、日志分析等运维工作，成本会显著提升。在服务划分之前，应该首先构建自动化的工具及环境，开发人员应该以自动化为驱动力，简化服务在创建、开发、测试、部署、运维上的重复性工作。任何重复性的工作都应该自动化。当代码提交后，自动化的工具链自动编译、构建、测试代码，验证代码是否合法，是否满足统一标准，是否存在安全漏洞。开发人员持续优化代码，当满足上线要求的时候，自动化部署到生产环境，这种自动化的方式，能够实现更可靠的操作，既避免了人为失误，又避免了微服务数量增多带来的开发、管理复杂化。

只有真正拥抱自动化的时候，才能做到持续发布，才能做到更好的用户体验。

## 演进式设计原则

我曾经经历过这样的一个失败案例：当时设计一个网站，负责人要求我们前期做大量的架构设计，考虑很多两年后的功能，并要进行大量的论证。但是当网站上线的时候，才发现很多客户的需求并不是我们最初设计的，虽然我们最初做了很多可用性相关的设计，仍然出现了很多故障，导致可用性非常低。

架构是持续演进的，并非一蹴而就的。单凭设计阶段很难达到理想的目标，需要不断锤炼。理想主义者会认为架构设计一旦完成，只要按照方案执行，就会设计出满意的软件。但是在软件开发过程中会遇到各种各样的问题，有可能会否定之前的某个决策，导致一系列变化。初级阶段应该采用尽可能简单的架构，因为初级阶段对需求、规模等都不是十分确定，可以采用快速迭代的方式进行架构演进。很多互联网公司都强调架构演进，如腾讯的一条重要发展原则就是“小步快跑”。

# 2

## 第 2 章 微服务架构

微服务架构是 Cloud Native 的重要组成部分。微服务架构给我们带来收益的同时，也会带来副作用，我们应该在什么阶段采用微服务架构？如何拆分微服务架构？拆分粒度多大比较合适？本章内容从问题开始，循序渐进，带领读者逐步深入微服务架构的各个角落。

### 2.1 微服务架构的起源

2005 年，Peter Rodgers 博士在云端运算博览会上提出微 Web 服务 (Micro-Web-Service)，将程序设计成细粒度的服务 (Granular Service)，以作为 Microsoft 下一阶段的软件架构。其核心思想是让服务按照类似 Unix 管道的存取方式使用，而且复杂的服务背后使用简单 URI 来开放界面，任何服务都能被开放 (exposed)。这个设计在 HP 的实验室被实现，具有改变复杂软件系统的强大力量。

2014 年，Martin Fowler 与 James Lewis 共同提出了微服务的概念，定义了微服务架构是以开发一组小型服务的方式来开发一个独立的应用系统，每个服务都以一个独立进程的方式运行，每个服务与其他服务使用轻量级 (通常是 HTTP API) 通信机制。这些服务是围绕业务功能构建的，可以通过全自动部署机制独立部署，同时服务会使用最小规模的集中管理 (例如 Docker) 能力，也可以采用不同的编程语言和数据库。

实际上，微服务的诞生绝非偶然，敏捷开发帮助我们减少浪费、快速反馈，以用户体验为目标；持续交付促使我们更快、更可靠、更频繁地改进软件；基础设施即代码（Infrastructure As Code）帮助我们简化环境的管理，这些都是推动微服务诞生的重要因素。如果没有这些基础，微服务架构在展现魅力的同时，可能由于各种问题导致最终失败。

从 SOA 架构到服务化架构，再到微服务架构，是一个逐步演进的过程。Amazon 被认为是微服务的鼻祖。2015 年我曾经接触过一个 Amazon 的工程师，他并不是特别了解微服务这个名词，直到看完 Martin Fowler 关于微服务的文章，才发现自己一直在做的就是微服务架构。可以说微服务架构并不是什么技术创新，而是开发过程发展到一定阶段对技术架构的要求，是在实践中不断摸索而来的。每个公司所信奉的架构思想有相同之处，但是也不尽相同。这种化繁为简的拆分方式，不只在技术上带来突破，更带来了很多潜在的价值，如关注点分离、沟通效率提升、快速演进、快速交付、快速反馈等。

## 2.2 为什么采用微服务架构

### 2.2.1 单体架构与微服务架构

就像很难用一个绝对的方式去判断架构好坏一样，在大多数场景下，我们也很难从一个外部的视角去判断服务拆分粒度的合理性，需要对上下文非常了解才能做出一个好决策。例如，团队规模多大，代码规模多大，有没有平台化，有没有工具链，是否需要持续交付，团队文化如何等。因此，一个外部的架构师是很难在短时间内将架构规划合理的，这需要一个过程，当真正了解这一切之后，不断权衡才能最终确定。在规划之前，有必要参考表 2-1，综合各方面的情况，最终做出决策。

表 2-1 单体架构与微服务架构对比

因 素	单体架构	微服务架构	说 明
交付速度	较慢	较快	服务拆分后，各个服务可以独立并行开发、测试、部署，交付效率提升，产品的更新速度会更快，用户体验更好。代码规模越大，微服务的优势越明显
故障隔离范围	线程级	进程级	服务独立运行，通过进程的方式隔离，使故障范围得到有效控制、架构变得更简单可靠。根据业务的重要程度划分服务，把核心的业务划分为独立的服务，这样从数据库到服务可以保持有效的故障隔离，进而保持稳定
整体可用性	较低	更高	微服务架构由于故障范围得到有效隔离，整体可用性更高，降低一点故障对整体的影响



续表

因 素	单体架构	微服务架构	说 明
架构持续演进	困难	简单	由于微服务的粒度更小,架构演进的影响面就更小。不存在大规模重构导致的各种问题。微服务架构对架构演进更友好
沟通效率	低	高	业界普遍认为团队规模越大,沟通效率越低,微服务架构按业务构建全功能团队,把权力下放,不会出现决策瓶颈点,缩小了沟通规模,提高了沟通效率
技术栈选择	受限	灵活	如果某个业务需要独立的技术栈,可以通过服务划分、接口集成的方式实现。例如搜索的技术栈、专业细分领域都不相同,通常采用独立的服务实现
可扩展性	受限	灵活	微服务架构可以根据服务对资源的要求以服务为粒度扩展,符合 AKF 扩展立方体中的 Y 轴扩展,而单体架构只能整体扩展,只能做到 AKF 扩展立方体中的 X 轴扩展
可重用性	低	高	微服务架构可以实现以服务为粒度通过接口共享重用
实现业务复杂性分解难度	困难	容易	微服务架构通过将业务分解为更多的服务,业务边界更清晰,更容易把一个复杂的问题分解为简单的小问题
产品创新复杂度	困难	容易	微服务架构以服务为粒度独立演进,团队有更多的自主决策权,更多的试错机会,更利于创新
一致性实现成本	低	高	服务划分后,如果服务 A 同时调用服务 B 和服务 C,如何保证同时成功或失败?单体架构下的单库事务变成了分布式事务问题
时延	低	高	服务划分后,调用次数增加,导致响应时间增加、吞吐量降低,如何弥补
资源成本	低	高	吞吐量的下降意味着要增加更多的资源,对于交付型项目,特别是小规模部署的场景下,是比较致命的
关联查询复杂度	简单	复杂	微服务架构的一个非常明显的特征就是,一个服务所拥有的数据只能通过这个服务的 API 来访问。通过这种方式来解耦,往往会带来查询问题。以前通过 join 就可以满足要求,现在如果需要跨多个服务集成查询,就会非常麻烦
远程调用	不涉及	涉及	微服务存在更多的远程调用,需要额外考虑序列化、通信协议、数据压缩、服务间的负载均衡、容错等问题
服务治理	不涉及	涉及	由于服务数量变多,微服务架构需要额外考虑服务的注册发现、依赖关系、治理等问题
对开发人员的要求	低	高	微服务架构更复杂,开发人员端到端负责,既要考虑接口定义,又要考虑数据库设计,对开发人员的水平要求更高
对工具的依赖	较低	较高	微服务架构中服务的数量较多,使用工具的效果更明显,依赖程度更高
运维复杂度	低	高	微服务架构中服务的数量较多,对服务的监控、健康检查要求更高,整体运维复杂度更高



## 2.2.2 什么时候开始微服务架构

产品初期优先选择单体架构。面对一个新的领域，对业务的理解很难在开始阶段就比较清晰，往往是经过一段时间之后，才能逐步弄清楚。很多时候，从一个已有的单体架构中逐步划分服务，要比一开始就构建微服务简单得多。另外，在资源受限的情况下，采用微服务架构风险较大，很多优势无法体现，性能上的劣势反而会比较明显。

单体、组件化、微服务架构成本趋势<sup>①</sup>，如图 2-1 所示。当业务复杂度达到一定程度后，微服务架构消耗的成本才会体现优势，并不是所有的场景都适合采用微服务架构，服务的划分应逐步进行，持续演进。产品初期业务复杂度不高的时候，应该尽量采用单体架构。

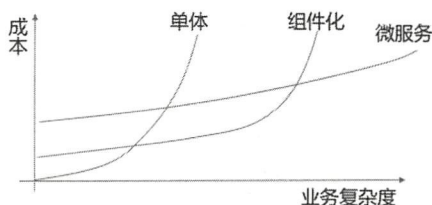


图 2-1 单体、组件化、微服务架构成本趋势

摘自 Martin Fowler 博客<sup>②</sup>的内容简单翻译如下。

当我听到关于使用微服务架构的故事的时候，我注意到了一种通用的模式。

1. 几乎所有成功的微服务架构都是从一个巨大的单体架构开始的，并且都是由于单体架构太大而被拆分为微服务架构。

2. 几乎在所有我听说过从一开始就构建为微服务架构的故事中，最终都有人遇到了巨大的麻烦。

在服务划分之前，应该保证基础设施及公共基础服务已经准备完毕。可以通过监控快速定位故障，通过工具自动化部署、管理服务，通过服务化框架降低服务开发的复杂度，通过灰度发布提升可用性，通过资源调度服务快速申请、释放资源，通过弹性伸缩快速扩展应用。

## 2.2.3 如何决定微服务架构的拆分粒度

微服务架构中的微字，并不代表足够小，应该解释为合适。但是“合适”过于含糊，

<sup>①</sup> 观点参考 <https://www.martinfowler.com/bliki/MicroservicePremium.html>。

<sup>②</sup> 来自 <https://martinfowler.com/bliki/MonolithFirst.html>。

每个人理解的合适都不尽相同。实际上，有时候对于一个对业务理解不够深入，对团队情况又不是很了解的人，根本无权协助确定服务的粒度。况且，就算本团队的架构师，也很难确定粒度。随着业务发展，开发人员水平的提升，粒度可能会发生变化。这是一个磨合的过程，一个不断演进的过程，没有绝对的对与错。

如果实在找不到合适的依据，可以参考表 2-2。决策占比是从通用的角度考虑，并不适用所有的情况，某些公司认为团队规模是决定性的，也有些公司认为架构演进是决定性的，还有些公司认为交付速度是决定性的，找到那个你认为的决定性因素，去做合理的拆分即可。

表 2-2 微服务拆分粒度决策参考表

因 素	决策占比	说 明
团队规模	50%	团队规模变大会出现决策瓶颈点，即所有的决策都要依赖于某个会议或某个人，没有人愿意承担责任，效率十分低下
交付速度要求	30%	毫无疑问，拆分粒度越小，交付时受到的约束越小，速度越快
其他	20%	例如，对占用资源的要求、对性能的要求、对一致性的要求、对架构演进速度的要求、对创新速度的要求

## 2.3 微服务设计原则

在微服务架构的设计过程中，我们应该遵循哪些原则？以下原则在微服务架构中经常被提起，遵循这些原则能够让我们少走弯路。

### 垂直划分优先原则

应该根据业务领域对服务进行垂直划分，因为水平划分服务可能会导致如下问题。

- 调用次数更多，导致性能大幅下降。
- 实现一个功能要跨越更多服务，沟通成本增加。

垂直划分服务可以以最简单的方式缓解上述问题，并且可以让团队从上至下关注业务实现，端到端负责，持续改进。图 2-2 简单描述了一个按业务领域垂直划分的微服务架构示例，在业务垂直方向切分服务，通过 API Gateway 聚合内容。

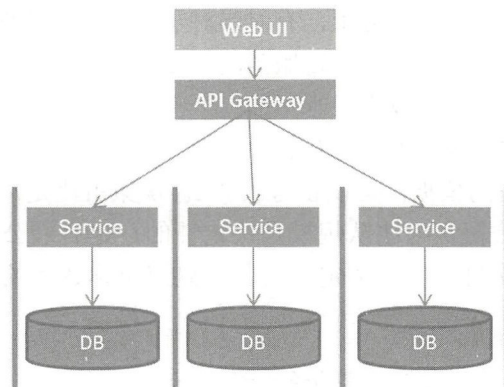


图 2-2 垂直划分服务

## 持续演进原则

服务数量快速增长带来架构复杂度急剧升高，开发、测试、运维等环节很难快速适应，会导致故障率大幅增加，可用性降低。非必要情况，应逐步划分、持续演进，避免服务数量的爆炸性增长。这等同于灰度发布的效果，先拿出几个不太重要的功能拆分出一个服务做试验，如果出现故障，则可以减少故障的影响范围。另外，除了业务服务数量的增加，还需要准备持续交付的工具、微服务框架等，并加强监控。

## 服务自治、接口隔离原则

尽量消除对其他服务的强依赖，这样可以降低沟通成本，提升服务稳定性。服务通过标准的接口隔离，隐藏内部实现细节。这使得服务可以独立开发、测试、部署、运行，以服务为单位持续交付。

直接访问对方的数据库会造成一定的耦合性，应该尽量避免。

## 自动化驱动原则

部署与运维的成本会随着服务的增多呈指数级增长，每个服务都需要部署、监控、日志分析等运维工作，成本会显著提升。在服务划分之前，应该首先构建自动化的工具及环境。开发人员应该以自动化为驱动力，简化服务在创建、开发、测试、部署、运维上的重复性工作，通过工具实现更可靠的操作。避免微服务数量增多带来的开发、管理复杂度问题。自动化可以从多个方面节省时间、提高效率，它可以快速跟踪整个交付过程并实时向所有参与者报告这个过程，赋予参与者责任感和成就感。如在研发过程中，推行持续集成

的文化就特别重要，而持续集成所依赖的工具就是一种自动化的体现。

很多互联网公司都遵循“一切皆自动化”的原则，特别是存在跨地域的研发模式时，使用自动化工具将是至关重要的，如开源的协作模式。

## 2.4 微服务架构实施的先决条件

不提倡从一开始就建立微服务架构的原因之一是没有做好准备，下面我们来看一下建立微服务架构前，需要从哪些方面做准备。

### 2.4.1 研发环境和流程上的转变

在实施微服务架构之前，我们要准备相关的环境和流程，可以简单地通过以下几个方面建立基本的条件。

#### 自动化工具链

微服务架构的一大优势是快速交付，快速交付不止体现在服务的粒度更小，可以独立交付，还体现在整个流程更快速。微服务架构基于自动化的工具链，以流水线交付的方式串联整个 DevOps 流程。小团队可以基于服务独立开发、测试、部署、运维。传统的交付周期以月为单位，而微服务架构的交付周期能做到以天为单位，按照传统的开发模式是无法满足这样的交付周期要求的。

#### 微服务框架

微服务框架可以封装、抽象分布式场景下的一些常用能力，例如负载均衡、服务注册发现、容错、远程通信等能力，可以让开发人员快速开发出高质量的服务，在采用微服务架构之前，应该先进行微服务框架的选型和试用。

#### 快速申请资源

如果以天为单位进行交付，就必须能够快速申请资源。基础设施即代码可通过编程的方式管理虚拟机或容器，免去了手动配置、更新各个硬件的环节，这就使得基础设施极具弹性，能够快速、高效、准确地进行重复性操作。开发人员使用同一套配置或代码，就可部署并管理成千上万台物理机。基础设施即代码能够得到更快的速度、更低的成本和更可靠的环境。用代码定义服务器配置意味着在众多服务器之间有绝对的一致性，容易形成标



准化。手动调整配置往往会有一些微妙的差异，难以追溯和调试，并且会导致许多诡异的问题。

## 故障发现反馈机制

当服务数量增多、交付频繁的时候，故障次数可能会大幅上升，我们需要通过全面的监控发现故障，及时处理并发出报警。当生产环境出现问题的时候，需要将故障进行分级，评估影响面，并分配给相应的架构师或者开发人员。开发人员需要不断更新故障的状态，以便管理者、客服、销售人员等问题相关人了解进度，来提供更好的用户体验。

## 研发流程上的转变

需要重新组建团队，以服务为核心，按照业务领域划分全功能团队，改变原有的研发流程、决策机制。例如，倡导敏捷文化、快速迭代，做更多的自动化测试，加强 Code Review，给团队更多的自主决策权等，具体内容可以参考本书第 9 章和第 10 章。

### 2.4.2 拆分前先做好解耦

解耦这个词来源于数学，是指使含有多个变量的数学方程变成能够用单个变量表示的方程组，即变量不再同时直接影响一个方程的结果，从而简化计算。

在软件世界里，解耦强调的是每个单元可以独立变化，尽量减少外界对系统内部的影响。说白了也就是，如果把 Memcache 换成 Redis，那么需要多少工作量，涉及的修改面有多大。解耦也会带来工作量的增加、架构或者代码变得复杂等问题。例如很多人会假设把 Oracle 换成 MySQL，Memcache 换成 Redis，但是在实际工作中，并不是所有的业务发展速度都有这么快，如果能预料到短期将发生变化，为什么不直接使用 MySQL 呢？通常这是一个伪命题。如果在未来几年后才发生变化，那么现在去做相应的适配，这不符合敏捷开发的哲学思想，也不是一个高效率的思路。

在转向微服务架构之前，业务服务存在状态、数据库中存在触发器和存储过程、服务之间绕过接口调用等问题，是我们首先要解决的。

## 状态外置

无状态（Statelessness）指的是服务内部变量值的存储。有状态的服务伸缩起来非常复杂，可以通过将服务的状态外置到数据库、分布式缓存中，使服务变成无状态。通常业界用牲畜来比喻无状态，用宠物来比喻有状态。宠物是需要呵护的，是有名字的，不能被轻



易替代的，而牲畜是没有名字的，只生产肉和奶，死掉一个用新的来替代即可。所以，我们期望服务可以做到无状态，可以被轻易地替代。

但是，无状态不代表状态消失，只是把状态转移到分布式缓存和数据库中了。业务服务伸缩的时候，还是要考虑分布式缓存和数据库所能承受的压力限制。那为什么还要外置呢？因为一方面即使不外置到数据库，数据库也存在状态，另一方面，这样可以把复杂度抽象到统一的位置，便于集中处理。例如，服务端的 Session 信息可以放到分布式缓存中，这一设计方法既可以让业务服务在一定范围内（分布式缓存的上限）伸缩时不受状态的限制，又可以把复杂度抽象到特定的位置，让专业领域开发人员统一做有状态的伸缩。虽然绝大多数服务都可以状态外置，但是并不是所有的业务服务都能设计成无状态，例如客户端与服务端的长连接，这种状态很难外置。

以下三种常见的状态需要和业务服务拆分开来，否则扩展性将受到很大限制。

- 定时任务：因为大多数任务不能重复触发，否则轻则重复做无用功（幂等的情况下），重则会导致不一致。例如从 A 表中把数据迁移到 B 表中，如果在两个服务中同时处理，没有一个协调器的话，会导致重复拉取。所以，需要把定时任务从业务服务中提取出来，通过分布式任务调度统一协调。
- 本地存储：在本地存储文件也是比较常见的。当有多个实例的时候，要么全部同步一遍，要么需要根据用户路由到同一个实例，并且在伸缩的过程中需要迁移。
- 本地缓存：某些业务会将数据存放在本地做缓存，例如 Session 数据。如果要去掉本地缓存，则可以通过分布式缓存和 Cookie 解决业务服务带状态的问题。当然，本地缓存也有适用的业务场景，不能一概而论。

## 去触发器、存储过程

触发器、存储过程在系统规模比较小的时候，的确非常简单实用。随着业务的发展，业务服务比较容易扩展，数据库通常变成了伸缩的瓶颈，许多方案都是为了减轻数据库的压力而提出的。触发器、存储过程可能会带来如下问题。

- 整体的伸缩受到数据库的限制，因为触发器、存储过程难以扩展。
- 当存在水平分表的时候，可能无法满足需求。
- 如果触发器、存储过程过多，则会导致运维复杂度升高。

解决方案通常是通过外部的业务服务或者定时任务替换触发器及存储过程。

## 通过接口隔离

直接访问其他服务的数据库，如图 2-3 所示。CRM 直接调用 OA 的数据库，没有通



过接口调用。对 CRM 进行微服务架构拆分之前，需要先理清系统的外部依赖关系，如果存在多个系统共享一个数据库，就会导致耦合，影响可用性和扩展性，进而可能出现如下问题。

- 当 CRM 中的数据结构发生变化的时候，OA 也要跟着变化，导致开发的过程互相依赖。
- 有可能在 CRM 进行的限流是没用的，因为 OA 没有通过 CRM 提供的接口进行调用。
- 假设随着业务的发展，需要在 CRM 的数据库上做缓存，可能存在多个地方要考虑缓存的问题。

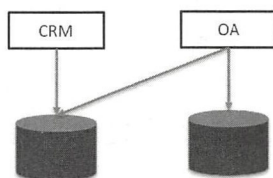


图 2-3 直接访问其他服务的数据库

总之，接口应该作为唯一对外提供的访问方式，这代表的是控制力。解决方法就是通过接口调用，逐步去除数据库的直接访问。

## 2.5 微服务划分模式

虽然服务是逐步被拆分出来的，随着业务的演进，在某一时刻，可能需要我们重新审视服务划分得是否合理。本节向大家推荐两种服务划分的方法，首先介绍如何选择服务划分的方法。

### 2.5.1 基于业务复杂度选择服务划分方法<sup>①</sup>

根据业务复杂度划分服务，如图 2-4 所示。当业务复杂度足够高的时候，应该基于领域驱动划分服务，而领域驱动本身足够复杂，很多概念比较抽象，应用范围并不是特别广泛，所以当业务复杂度较低时，可以选择基于数据驱动划分服务。数据驱动更容易理解和上手。也就是说，除非业务复杂度非常高，否则应该优先以数据驱动划分服务。这里的业务复杂度专指业务逻辑，而非数据量、并发量等相关复杂度。

<sup>①</sup> 观点参考了 Martin Fowle 的 *Patterns of Enterprise Application Architecturer* 一书。



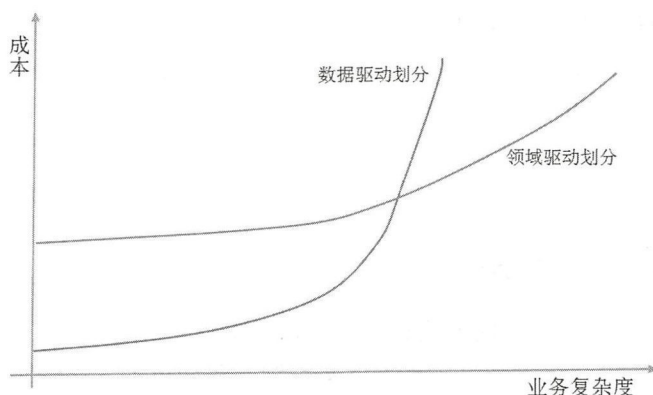


图 2-4 根据业务复杂度划分服务

在做出选择的时候，还有一个参考指标是，团队以前是否已经基于领域驱动开发业务。也就是说，如果产品已经基于领域驱动开发了一段时间，团队具备了领域驱动开发的能力，那么推荐继续选择领域驱动划分服务。如果是一个全新的产品，则可以灵活选择。

选择服务划分的方法时要重点考虑如下条件。

- 业务复杂度。
- 团队对领域驱动的熟悉程度。

### 2.5.2 基于数据驱动划分服务

数据驱动是一个自下而上的架构设计方法，数据驱动强调的是数据结构，也就是通过分析需求，确定整体数据结构，根据表之间的关系划分服务。

通常基于数据驱动划分服务的步骤如下。

(1) 需求分析。通过领域专家（或者产品经理）确定目标，然后总结 User Story，确定核心的业务流程；通过工具呈现比较粗糙的界面，进行内部讨论；不断迭代此环节，直到满意为止。

(2) 抽象数据结构。根据需求总结 Use Case，协助分析需求，从中抽象数据结构。

(3) 划分服务。分析数据结构，识别服务——服务应该满足高内聚、低耦合、单一职责等特征。

(4) 确定服务调用关系。先分析出主要流程，根据请求需要调用的服务确定服务调用关系。如果存在问题，则需要回到(1)重新开始。

(5) 业务流程验证。重新回到 User Story，以服务为粒度实现时序图，注意此阶段重点是验证服务划分是否合适，要关注如下问题。





- 一次更新操作如果要跨越更多服务，那么一致性的要求是什么。
- 跨服务查询时，是否要做关联查询，一个服务内是否能解决问题。
- 性能是否能满足要求。
- 成本是否能满足要求。

(6) 持续优化。

### 2.5.3 基于领域驱动划分服务

领域驱动<sup>①</sup>是一个自上而下的架构设计方法，通过和领域专家建立统一的语言，不断交流，确定关键业务场景，逐步确定边界上下文。领域驱动更强调业务实现效果，认为自下而上的设计可能会导致技术人员不能更好地理解业务方向，进而偏离业务目标。

通常基于领域驱动划分服务的步骤如下。

(1) 通过模型和领域专家建立统一语言。建立统一语言是为了更深入地理解需求。通用语言尽量以业务语言为主，而非技术语言；通用语言和代码一样，需要不断地重构。

(2) 业务分析。确定核心的业务流程，然后逐步扩展到全部。最好通过工具呈现比较粗糙的界面，供内部讨论。

(3) 寻找聚合。显式地定义领域模型的边界。最近比较热门的事件风暴<sup>②</sup>，是一种基于领域驱动分析业务、划分服务的方法。

事件风暴就是把所有的关键参与者都召集到一个很宽敞的屋子里来开会，并且使用便利贴来描述系统中发生的事情，如图 2-5 所示。

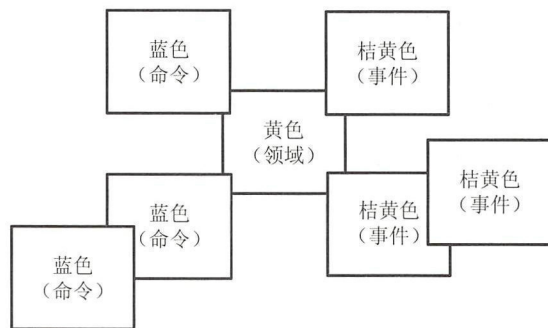
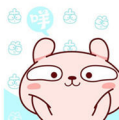


图 2-5 事件风暴

① 参考了 Eric Evans 的《领域驱动设计：软件核心复杂性应对之道》一书和 Vaughn Vernon 的《实现领域驱动设计》一书。

② 参考 [https://leanpub.com/introducing\\_eventstorming](https://leanpub.com/introducing_eventstorming)。



- 用桔黄色的便利贴代表领域事件，在上面用一句话描述曾经发生过什么事情。
- 用蓝色的便利贴代表命令。命令的发起者可能是人，也可能是注入系统中的外部事件，或者定时器等。
- 用黄色的便利贴代表聚合。聚合是一组相关领域对象的集合，高内聚、低耦合是其基本要求，聚合内还要保证数据一致性。

(4) 确定服务调用关系。先分析出主要流程，根据一次请求需要调用的服务来确定服务调用关系。如果存在水平划分，则需要根据服务依赖原则确定关系。如果存在问题，则需要回到(1)重新开始。

(5) 业务流程验证。以服务为粒度实现时序图，注意此阶段重点是要验证服务划分是否合适，主要关注如下问题。

- 一次更新操作如果要跨越更多服务，那么一致性的要求是什么。
- 跨服务查询时，是否要做关联查询，一个服务内是否能解决问题。
- 性能是否能满足要求。
- 成本是否满足要求。

(6) 持续优化。

## 2.5.4 从已有单体架构中逐步划分服务

在大多数场景下，并非从开始阶段就采用微服务架构，而是随着业务不断发展，从最初的单体架构中逐步拆分服务。下面描述了一个单体架构逐步拆分的步骤。

(1) 所有微服务成功的故事都是从一个单体架构太大，需要被拆散开始的，如图 2-6 所示。我们应该从单体架构开始，当系统规模足够大、团队人数足够多时，再逐步拆分服务，通常前后端分离是拆分的第一步。

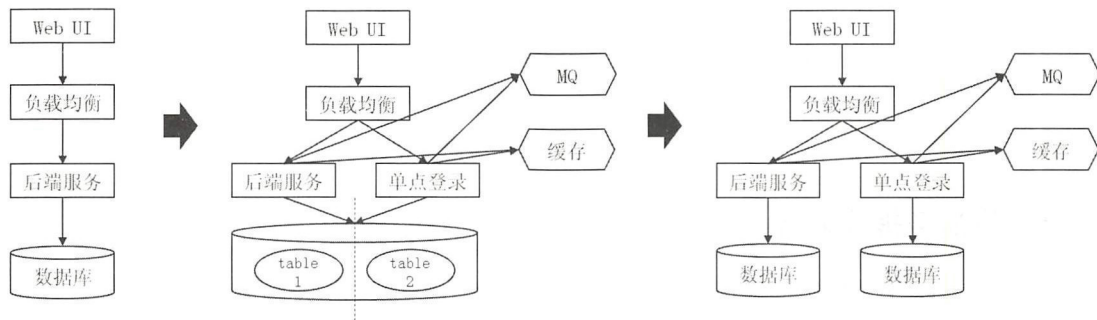


图 2-6 从已有架构逐步拆分服务（一）



(2) 提取公共基础服务，如单点登录。拆分可以遵循逻辑分离和物理分离两种方法<sup>①</sup>。另外随着系统压力的增加，可能会用到消息中间件、分布式缓存等服务。

(3) 不断地从老系统中抽象出服务，垂直划分优先，如图 2-7 所示。

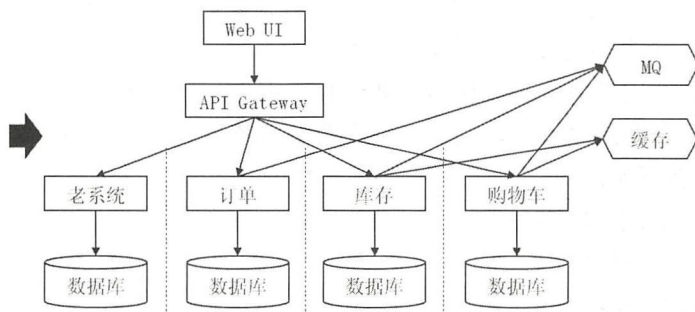


图 2-7 从已有架构逐步拆分服务（二）

(4) 当业务越来越复杂的时候，API Gateway 做了太多的事情，会成为一个瓶颈点，服务之间的依赖关系也会变得越来越复杂，此时，需要适当地进行水平切分，如图 2-8 所示。

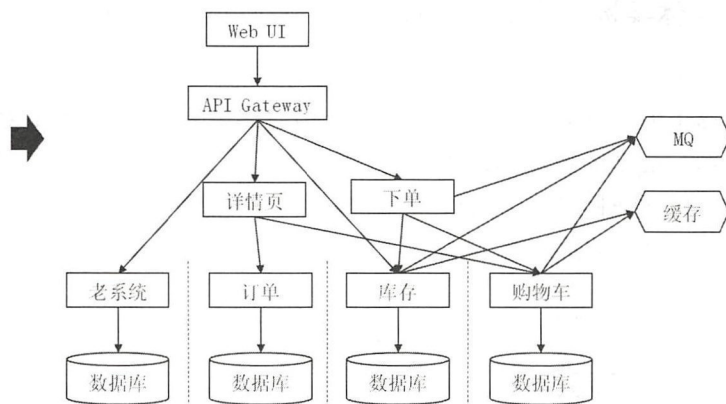
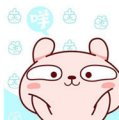


图 2-8 从已有架构逐步拆分服务（三）

## 2.5.5 微服务拆分策略

当不断从单体架构中抽象服务的时候，哪些服务优先被拆分，哪些服务不需要被拆分？以下几个策略可以帮助解决拆分中的这些问题。

<sup>①</sup> 可以参考第 2.4.2 节的介绍，拆分前先做好解耦。





- 比较独立的新业务优先采用微服务架构。从成本角度考虑，新业务采用新的架构是最合理的，因为这样做对老业务的影响最小。
- 优先抽象通用服务。因为通常通用服务的边界比较明显，耦合度低，比较容易分离。
- 优先抽象比较容易识别的、边界比较明显的服务。如果原有包结构比较清晰，可以基于原有包结构中有明显边界的、比较完整的业务进行划分，这是从成本角度考虑。如果已经基于单体架构开发了一段时间，对业务的理解程度已经非常高，那么开发及架构人员能够比较容易地提炼出一些边界比较明显的服务。
- 优先抽象核心服务。因为微服务的开发及运维成本比较高，并不是所有的地方都需要划分很小的粒度。往往一些比较边缘的运营、管理的系统甚至不会考虑拆分。另外，随着时间的推移，有一些业务可能会发生改变，因此应该先抽象出核心服务。
- 优先抽象具有独立属性的服务。应根据功能的变更频率、资源占用、技术栈等属性划分服务。
- 采用绞杀者模式，在遗留系统外围，随着时间的推移，让新的服务逐渐“绞杀”老的系统。在这种情况下，复杂度往往体现在如何灰度发布、迁移数据，以及如何保障服务不中断，后面的章节会详细描述。

### 2.5.6 如何衡量服务划分的合理性

每个产品在实施微服务架构最初的动力都不一样，目标也有所区别，所以判断是否划分合理，首先要看是否达成了目标。其次，可以参考以下几种衡量方式。每种衡量方式不能单独作为一个判断标准，需要综合考虑。

- 一个小功能的修改从需求到上线需要多长时间？正常情况下的微服务架构交付周期应该是以天为单位的。如果一个小功能的修改需要几周到几个月的时间，可能意味着服务划分粒度过大，存在太多的冲突，要等待合并代码。
- 大多数功能修改是否可以在一个服务内完成？如果经常需要跨服务团队的联合开发组才能完成一个新功能的开发或者旧功能的修改，则说明服务划分存在问题。
- 是否要频繁修改接口？频繁修改接口有可能是接口设计不合理导致的，也有可能是服务划分的问题导致的，说明服务之间的边界并不是特别明确和稳定。
- 响应时间是否能满足要求？在某些追求极致性能的场景中，对响应时间要求较高，服务划分的层次太多、粒度太小都可能导致响应时间不能满足要求。
- 是否存在大量的跨服务更新？是否存在大量的跨服务的关联查询？出现这两个问题，可能是因为划分不合理。







## 2.6 微服务划分反模式

前面我们介绍了如何划分服务，在此之上，我们希望通过微服务划分的反模式来帮助大家少走弯路。

### 根据代码行数划分服务

代码规模太大会导致沟通效率、交付效率低下，耦合度高，以及比较笨重。代码规模可以作为一个参考，但是不能作为一个绝对标准，微服务架构中存在一个“大服务”是很正常的。基于代码行数拆分服务很难衡量服务的完整性，容易导向更小的拆分粒度，引起不必要的复杂度。

### 划分粒度越小越好

服务的大小并不是特别重要，可以根据团队规模、代码规模、业务复杂度、技术领域、重要程度、成本等因素综合考虑。关于服务粒度的大小，业界并没有统一标准，也很难衡量，最接近的衡量标准是研发团队规模。粒度小意味着更高的维护成本。后端管理、辅助系统通常粒度较大。

### 一次性划分服务

拆分服务有如下两种方式。

第一种，先拆分业务代码再拆分数据库。如图 2-9 所示，数据库并没有拆分，只是从单体中抽象出部分服务。

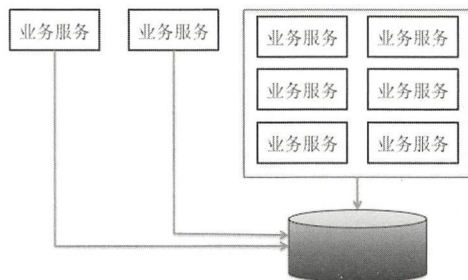


图 2-9 先拆分业务代码再拆分数数据库

第二种，业务代码和数据库同步拆分。如图 2-10 所示，部分业务服务被拆分出来的同时，数据库也被同步拆分出来。



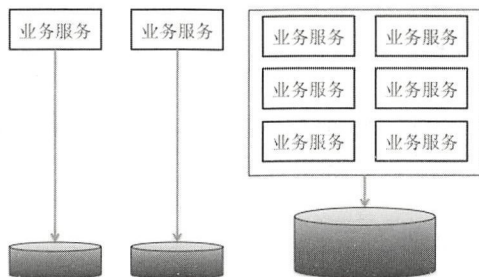


图 2-10 业务代码和数据库同步拆分

采用第二种方式拆分服务时，根据服务将数据库彻底拆分为多个独立的数据库，每个服务独享数据库服务，服务之间只能通过接口调用。这看起来非常美好，但需要为此做大量的数据迁移。当业务处于初期，需求不是非常确定，开发人员对业务理解不是特别透彻的时候，可能拆分后发现拆分得并不合理，只能再进行合并，又要进行一次数据迁移。相对来说，业务代码拆分成本更低，而数据迁移的成本更高，频繁、大量的数据迁移并不可取。

更好的做法是把第一种方式作为过渡阶段，当业务逐步稳定后再彻底进行数据迁移。注意，处于过渡阶段时数据库并没有彻底分离，一切依赖都通过接口访问。但是这只是口头上的约定，对于业务开发人员直接在数据库中进行关联查询。需要通过 Code Review 的方式避免。

服务划分是一个长期的过程，需要积累大量的领域知识，以此来理解核心流程。最好是和领域专家一起组成联合团队，在理解核心问题的情况下，持续拆分服务并验证拆分合理性，随着时间的推移，还可以重新划分。可见，服务拆分是一个持续性的过程。

## 服务划分一旦完成，不能改变

由于业务的不断变化，以及开发人员对领域知识和其他影响因素的理解等问题，很难一次性做出一个完美的解决方案。通常在划分后会发现，某个问题是不可忍受的，例如划分后导致响应时间降低，增加了更多的成本，有可能需要重新合并服务；由于业务的变化，原本的依赖关系发生了变化，有可能面临需要重新划分服务等类似的问题。

## 先实施组件化，再实施微服务架构

很多技术人员试图先通过组件化逐步过渡到微服务架构，这是一种错误的思路。微服务架构划分更强调业务领域的完整性，因此垂直划分优先，而组件化往往通过抽象出稳定的部分形成组件共享，对调用次数和依赖关系并不强调。因此，组件化之后再转化到微服务架构的方式，通常是错误的。



## 2.7 微服务 API 设计

API 是服务之间通信的契约，通过 API 可以忽略服务内部实现的细节。如果接口设计良好，则可以降低团队间的耦合度，加快开发速度。

### 2.7.1 优秀 API 的设计原则

设计出优秀的 API 通常需要遵循如下原则。

- 简单：API 应该符合大多数人正常的思维逻辑，避免异想天开的交互，满足需求的同时，越简单越好。
- 易懂：优秀的 API 可读性好，尽量做到不需要文档就能读懂接口名称、参数的大概含义，提供给第三方开发者的接口要进行详细地描述，包括参数的取值范围、错误码、异常返回规则、SLA 相关指标等。
- 一致：对于同一个公司、站点提供的 API，最好有统一的规则，让开发者只要看过几个 API 之后，基本都能猜到剩余 API 的含义。
- 稳定：最好在开始的时候就考虑好，不要轻易修改 API，否则会给使用者造成非常大的麻烦。如果修改 API 无法避免，那么最好通过增加接口而不是修改已有接口做到兼容。如果一定要改变已有接口，也要通过明确的版本号加以区分，并且预留足够的时间。
- 安全：设计时要考虑超出预期的情况如何处理，给出被限流的情况。下面是 GitHub API 使用的两个相关的头部参数。
  - X-RateLimit-Limit：每小时允许发送请求的最大值。
  - X-RateLimit-Remaining：当前时间窗口剩下的可用请求数目。
- 如果提供给第三方，则要考虑如何认证，租户之间如何隔离，并且尽量使用 HTTPS 协议。如果验证失败，则要返回 401 Unauthorized 状态码；如果没有被授权访问，则要返回 403 Forbidden 状态码，并且提供详细的错误信息。

### 2.7.2 服务间通信——RPC

RPC (Remote Procedure Call) 是指远程过程调用。简单来说，RPC 希望达到的效果是，调用远程方法就像调用本地方法一样。由于调用方和被调用方实际上分布在不同的机器上，一般情况下，会有一个框架来支撑，以便屏蔽底层通信细节，让双方开发起来更简单。

一个普通的 RPC 调用流程如图 2-11 所示，具体调用过程如下。

(1) 客户端在业务服务中发起请求。



(2) 调用本地 stub<sup>①</sup>，本地 stub 对消息进行序列化、封装等处理。

(3) 客户端 stub 调用网络通信模块将消息送达服务端，中间还可能包括寻址、建连接等一系列操作。

(4) 服务端网络通信模块把接收到的消息转发给 stub 进行反序列化。

(5) stub 转发给业务服务进行处理并返回结果。

(6) stub 将结果进行序列化，调用网络通信模块。

(7) 服务端通过网络通信模块将结果返回到客户端网络通信模块中。

(8) 客户端网络通信模块将消息转发给 stub。

(9) stub 进行反序列化并转发给客户端业务服务。

(10) 客户端得到最终结果。

我们的目标是让 (2) 到 (9) 对开发者不可见。

既然要求像调用本地方法一样调用远程方法，那就不只是要屏蔽复杂度，还要在性能上进行考虑。

影响 RPC 性能的因素如下。

- 序列化。常用的 RPC 序列化协议包括：Thrift、Protobuf、Avro、Kryo、MsgPack、Hessian、Jackson。

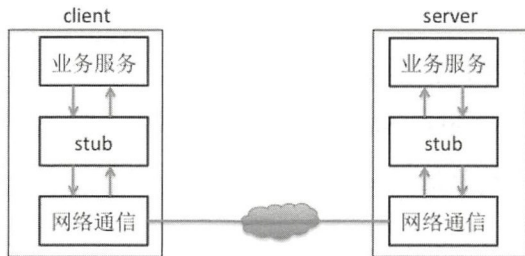
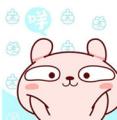


图 2-11 RPC 调用流程图

- 传输协议。常用的传输协议包括：HTTP、Socket、TCP、UDP 等。
- 连接。连接包括：长连接、短连接。
- IO 模型。常用的网络 IO 模型：同步阻塞 IO（Blocking IO）、同步非阻塞 IO（Non-blocking IO）、IO 多路复用（IO Multiplexing）、异步 IO（Asynchronous IO）。

① 在这里 stub 一般被翻译为存根，即在本地存在一个和远程一样的方法。在 Java 中通常通过依赖对方提供的接口来实现。例如服务端将对外提供的接口打包成 JAR 文件，客户端要依赖于服务端提供的 JAR 包，这个 JAR 包就可以理解为存根。





### 2.7.3 序列化——Protobuf

Protobuf 是由 Google 开源的消息传输协议，用于将结构化的数据序列化、反序列化通过网络进行传输，目前被广泛应用的主要版本有 Protobuf 2 和 Protobuf 3。Protobuf 首先解决的是如何在不同语言之间传递数据的问题，目前支持的编程语言有 Java、C++、Python、Ruby、PHP、Go 等。通过 Protobuf 可以很容易地实现一个 Client 和一个 Server 之间的数据传递，Client 和 Server 可以使用不同语言。

Protobuf 是一个高性能、易扩展的序列化框架，通常是 RPC 调用追求高性能的首选。它本身非常简单，易于开发，结合 Netty 可以非常便捷地实现 RPC 调用。同时，可以利用 Netty 为 Protobuf 解决有关 Socket 通信中“半包、粘包”等问题（反序列化时，字节成帧）。

Protobuf 比 JSON、XML 等更快、更轻、更小，并且可以跨平台。Protobuf 最新的版本为 3.x，如果使用它，首先要编写 proto 文件，即 IDL 文件（后缀为“.proto”的文本文件），然后通过客户端生成 Java 相关类进行序列化、反序列化。

下面我们就简单描述一下基于 Protobuf 实现序列化、反序列化的步骤。

#### 1. 安装客户端

通过 brew 可以实现在 Mac 下安装 Protobuf，执行命令如下。

```
brew install protobuf
```

如果出现如下执行结果，意味着已经安装成功了。

```
/usr/local/Cellar/protobuf/3.5.1: 267 files, 18.0MB
```

也可以通过如下命令检验是否安装成功。

```
protoc --version
```

#### 2. 安装工具

为了在编辑 proto 文件的时候获得更好的体验，可以先安装插件。idea 里可以安装 Protobuf Support，在插件库中搜索后直接安装即可，如图 2-12 所示。

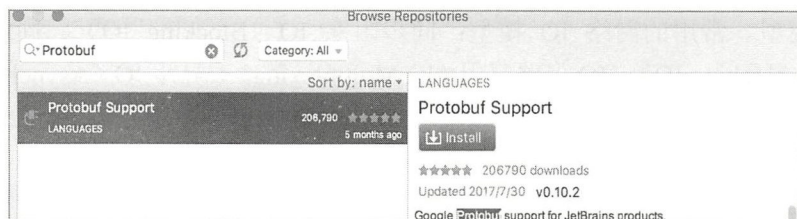


图 2-12 idea 安装 Protobuf Support 插件



Protobuf Support 可以对语法高亮显示、错误提示, 如图 2-13 所示。

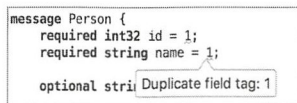


图 2-13 Protobuf Support 插件效果

### 3. 编辑 proto 文件

本节采用的 Protobuf 3.x, 需要遵循 proto 3 的语法。关于 Protobuf 的数据类型和 Java 的数据类型如何对应, 可以参照表 2-3 提供的对应关系。

表 2-3 Protobuf 的数据类型和 Java 的数据类型的对照表<sup>①</sup>

.proto	C++	Java	Python	Go	Ruby	C#
double	double	double	float	float64	Float	double
float	float	float	float	float32	Float	float
int32	int32	int	int	int32	Fixnum or Bignum	int
int64	int64	long	int/long <sup>[3]</sup>	int64	Bignum	long
uint32	uint32	int <sup>[1]</sup>	int/long <sup>[3]</sup>	uint32	Fixnum or Bignum	uint
uint64	uint64	long <sup>[1]</sup>	int/long <sup>[3]</sup>	uint64	Bignum	ulong
sint32	int32	int	intj	int32	Fixnum or Bignum	int
sint64	int64	long	int/long <sup>[3]</sup>	int64	Bignum	long
fixed32	uint32	int <sup>[1]</sup>	int	uint32	Fixnum or Bignum	uint
fixed64	uint64	long <sup>[1]</sup>	int/long <sup>[3]</sup>	uint64	Bignum	ulong
sfixed32	int32	int	int	int32	Fixnum or Bignum	int
sfixed64	int64	long	int/long <sup>[3]</sup>	int64	Bignum	long
bool	bool	boolean	boolean	bool	TrueClass/FalseClass	bool
string	string	String	str/unicode <sup>[4]</sup>	string	String(UTF-8)	string
bytes	string	ByteString	str	[]byte	String(ASCII-8BIT)	ByteString

定义 product.proto 文件的过程如下。

```
syntax = "proto3"; // 声明支持的版本是 proto 3

option java_package = "com.cloudnative.protobuf"; // 声明包名, 可选
option java_outer_classname = "ProductProtos"; // 声明类名, 可选
```

① 来自 Protobuf 官网 <https://developers.google.com/protocol-buffers/docs/proto>。

```
message Product {  
  
    int32 id = 1;  
    string name = 2;  
    string price = 3;  
  
    enum ColorType { //定义枚举类型  
        WHITE = 0;  
        RED = 1;  
        BLACK = 2;  
    }  
}
```

在命令行执行如下代码。

```
protoc --java_out=../java product.proto
```

--java\_out 表示在目标目录中生成 Java 代码。由于已将 product.proto 放到 src/main/java/proto 目录下，所以--java\_out=../java 参数会将生成的 Java 类创建到 java 目录下，如图 2-14 所示。

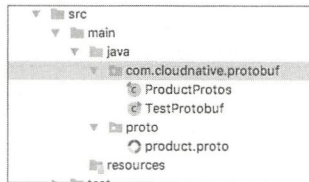


图 2-14 项目目录结构

打开 ProtobufProtos 会发现，这是一个非常复杂的类，其代码大概 1500 行。

#### 4. 使用 Protobuf 序列化

引入 Protobuf-java 包，可以先通过 <http://mvnrepository.com> 查询，然后点击复制，非常方便，如图 2-15 所示。

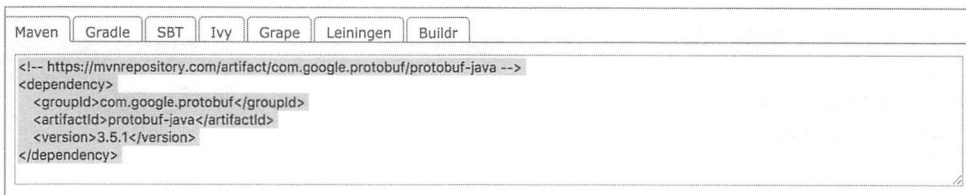


图 2-15 查询结果



实际上, Protobuf 的序列化及反序列化非常简单。Protobuf 生成的类中已经实现了相应的方法, 调用即可。示例代码如下。

```
package com.cloudnative.protobuf;

import com.google.protobuf.InvalidProtocolBufferException;

public class TestProtobuf {
    public static void main(String[] args){
        TestProtobuf testProtobuf =new TestProtobuf();
        byte[] buf=testProtobuf.toByte();
        try {
            ProductProtos.Product product = testProtobuf.toProduct(buf);
            System.out.println(product);
        } catch (InvalidProtocolBufferException e) {
            e.printStackTrace();
        }
    }
    //序列化
    private byte[] toByte(){
        ProductProtos.Product.Builder
productBuilder=ProductProtos.Product.newBuilder();
        productBuilder.setId(11);
        productBuilder.setName("milk");
        productBuilder.setPrice("4.12");
        ProductProtos.Product product =productBuilder.build();
        byte[] buf = product.toByteArray();
        System.out.println(buf.length);
        return buf;
    }
    //反序列化
    private ProductProtos.Product toProduct(byte[] buf) throws
InvalidProtocolBufferException {
        return ProductProtos.Product.parseFrom(buf);
    }
}
```

### 2.7.4 服务间通信——RESTful

REST 是 Roy Thomas Fielding<sup>①</sup>在 2000 年的博士论文<sup>②</sup>中提出的。REST 是

① Roy Thomas Fielding 是 HTTP 协议 (1.0 版和 1.1 版) 的主要设计者、Apache 服务器软件的作者之一、Apache 基金会的第一任主席。

② 论文地址 <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>。



Representational State Transfer 的缩写，通常翻译为“表现层状态转化”。如果一个架构符合 REST 原则，就称它为 RESTful 架构。

API 如何设计才能满足 RESTful 的要求呢？

### 1. 协议

API 是基于 HTTP 的协议。

### 2. 域名

API 要有一个域名，例如 `http://api.xxx.com`。

### 3. 版本

API 要有版本信息。当客户端数量较多或者提供给第三方使用时，很难控制客户端的兼容性，一个比较好的做法就是当已发布的服务变更时，通过版本来控制兼容性。当然，版本不能演进太快，最好的版本化就是无须版本，例如 `http://api.xxx.com/v1/`。

### 4. 路径

要理解 REST，首先要理解什么是资源（Resource）。REST 开发又被称作面向资源的开发。API 要用资源来表示，URL 中不能出现动词。资源是服务端可命名的一个抽象的概念，只要客户端容易理解，可以随意抽象。通常可以把资源看成是一个实体，例如用户、邮件、图片等，用 URI（统一资源定位符）指向它。经验告诉我们，往往这里的资源和数据库的表名是对应关系。一种观点认为 DDD 可以和 REST API 很好地契合，因为 REST 的资源可以很好地与 DDD 的实体映射起来。定义资源的时候，推荐用复数，假设我们要获取用户的信息，大概是这样：`http://api.xxx.com/v1/users/`。

### 5. 方法

一般允许的方法主要包括如下几种。

- GET：读取资源，一个或多个（常用）。
- POST：创建资源（常用）。
- PUT：修改资源，客户端提供修改后的完整资源（常用）。
- PATCH：对已知资源进行局部更新，客户端只需要提供改变的属性。
- DELETE：删除、回收资源（常用）。
- HEAD：读取资源的元数据（不常用）。
- OPTIONS：读取对资源的访问权限（不常用）。

一般情况下，GET、POST、PUT、DELETE 已经足够，甚至有一种观点认为，只需要

使用 GET、POST 即可，例子如下所示。

- GET/users/1，获取用户 ID 为 1 的用户信息。
- GET/users/1/orders，获取用户 ID 为 1 的用户拥有的所有订单。

## 6. 参数

参数可以放到 API 路径中，也可以放到“?”的后面。

```
GET/users/1/orders
GET/orders?user_id=1
```

## 7. 编码

虽然 RESTful 并没有限制资源的表达格式，HTML/XML/JSON/纯文本/图片/视频/音频等都可以，但是通常服务端和客户端通过 JSON 传递信息。

## 8. 状态码

用 HTTP Status Code 传递 Server 的状态信息，常用的状态码如下。

- 100 Continue
- 200 OK, GET
- 201 Created, POST/PUT/PATCH
- 202 Accepted
- 204 NO Content, DELETE
- 400 Bad Request, POST/PUT/PATCH
- 401 Unauthorized
- 403 Forbidden
- 404 Not Found
- 405 Method Not Allowed
- 406 Not Acceptable
- 409 Conflict
- 410 Gone
- 412 Precondition Failed
- 429 Too many requests
- 500 Internal Server Error
- 501 Not Implemented
- 503 Service Unavailable

完整信息可以参考 w3 官网<sup>①</sup>。

要理解 RESTful，还要考虑如下重要的约束条件。当然，这些条件也不是绝对的，需要结合业务场景来确定。

- 单一职责。尽量保持接口职责单一，留给客户端足够的操作空间，以满足不同的业务需求。对于接口粒度的大小，需要考虑的因素包括：性能（合并请求性能更高）、一致性、灵活性及客户端的易用程度。
- 幂等性。一次和多次请求某一个资源应该具有同样的作用，客户端能够重复发起请求而不必担心造成副作用。
- 无状态。多次请求之间不应该存在状态耦合，无须关联过去、现在和将来的请求或者响应。
- 客户端发起。一般通信方式都是由客户端发起的，服务端是被调用的。随着 HTTP/2 的到来，这一条可能会发生变化。
- 原子性。保证所有操作是一个不可分割的单元，要么全部成功，要么全部失败，需要结合业务要求加以确定。
- 易用。需要提供详尽的文档、参数说明、示例等，API 定义的 URL、变量名要通俗易懂，最好是英文，尽量减少自定义的缩写，让开发者容易调试和管理。
- SLA。需要提供响应时间、吞吐量、可用性等关键指标。

RESTful 已经成为业界的主流，主要是因为 RESTful 通常采用 HTTP+JSON 的方式实现，继承了 HTTP 和 JSON 的优点。相对于 SOAP、RPC 等方式，RESTful 更加轻量、简单，支持跨语言，并且容易调试。

### 2.7.5 通过 Swagger 实现 RESTful

传统的 API 设计通常先完成代码，然后另外补充一份说明文档，这种方式效率比较低，文档和代码缺乏关联性。更高级一点的做法是使用 JAVADOC，把文档和注释关联起来，以提升效率，但是由于 JAVADOC 需要不断生成，文档难免和代码存在不一致。

在此背景下，Swagger 诞生了。Swagger 是一个简单、功能强大、非常流行的 API 表达工具。基于 Swagger 生成 API，可以得到交互式文档、自动生成代码的 SDK，以及 API 的发现方式等。通过 Swagger 可以很容易地生成标准的 API，示例如图 2-16 所示。

---

<sup>①</sup> <https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>。

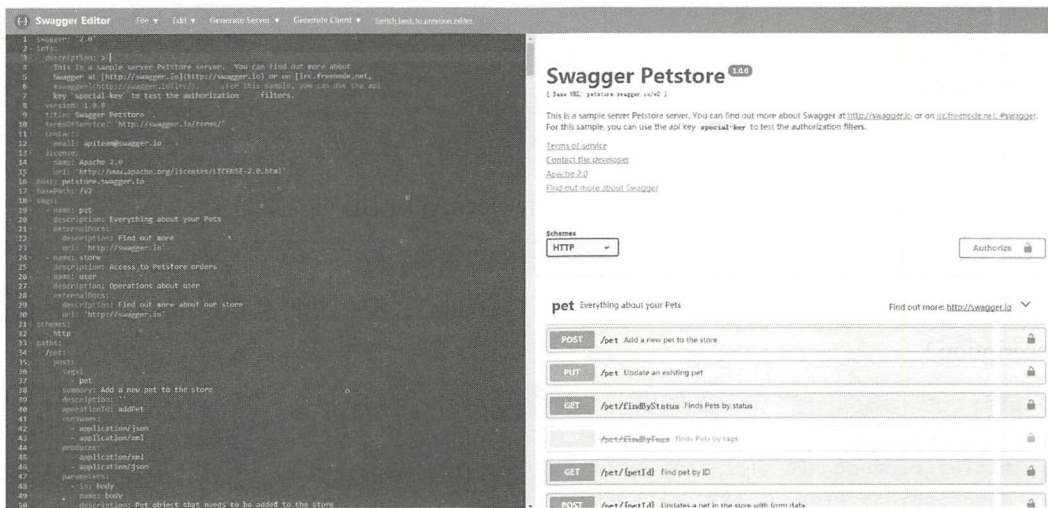


图 2-16 基于 Swagger 生成 API 的示例

Swagger 是基于 OpenAPI<sup>①</sup>的，OpenAPI 支持 YAML<sup>②</sup>和 JSON 格式描述 API。YAML 相对于 JSON 来说更加简洁，比较适合做简洁的序列化和配置文件。编写 YAML 文档推荐使用 Swagger Editor，它提供了语法高亮、自动完成、即时预览等功能。编辑器可以在本地使用，也可以在线使用。YAML 的数据结构可以用类似大纲的缩排方式呈现，结构通过缩进来表示，连续的项目通过“-”来表示，map 结构里面的 key/value 对用“:”来分隔。

基于 Swagger Editor 设计 API，可以直接在线编辑 API，也可以在本地上安装，下面以在线编辑器为例介绍如何基于 Swagger 构建 API。

- (1) 访问官方 Editor 编辑器 <http://editor2.swagger.io>。
- (2) 编辑 YAML 文件，如下所示。

```
swagger: "2.0"

info:
  version: 1.0.0
  title: Product API
  description: Product API for test

schemes:
```

① OpenAPI 规范是 Linux 基金会的一个项目，试图通过定义一种用来描述 API 格式或 API 定义的语言，来规范 RESTful 服务的开发过程。目前它的最新的版本是 3.0，Swagger 支持到 2.0。

② YAML 是 YAML Ain't a Markup Language（YAML 不是一种置标语言）的递归缩写。



```
- https
host: localhost
basePath: /product

paths: {}
```

下面简单说明一下这个文档。首先，显示 OpenAPI 使用的版本是 2.0。

```
swagger: "2.0"
```

API 的描述信息，包括如 API 文档版本（version）、API 文档名称（title）和描述信息（description）。

```
info:
  version: 1.0.0
  title: Product API
  description: Product API for test
```

下面的代码表示 API 的 URL，采用 HTTPS 协议，介绍了主机名（host）、根路径（basePath）。

```
schemes:
  - https
host: localhost
basePath: /product
```

下面我们来看一个稍复杂一点的示例。

```
swagger: '2.0'
info:
  version: '1.0'
  title: Swagger 构建 RESTful API
  host: 'localhost:8080'
  basePath: /
tags:
  - name: product-controller
    description: Product Controller
paths:
  /products:
    get:
      tags:
        - product-controller
      summary: 获取产品列表
      description: 获取产品列表
      operationId: getProductListUsingGET
```

```
consumes:
  - application/json
produces:
  - '*'
responses:
  '200':
    description: OK
    schema:
      type: array
      items:
        $ref: '#/definitions/Product'
  '401':
    description: Unauthorized
  '403':
    description: Forbidden
  '404':
    description: Not Found

/products/{id}:
  get:
    tags:
      - product-controller
    summary: 获取产品详细信息
    description: 根据 URL 的 id 来获取产品详细信息
    operationId: getProductUsingGET
    consumes:
      - application/json
    produces:
      - '*'
    parameters:
      - name: id
        in: path
        description: 产品 ID
        required: true
        type: integer
    responses:
      '200':
        description: OK
        schema:
          $ref: '#/definitions/Product'
      '401':
        description: Unauthorized
      '403':
```



```
        description: Forbidden
      '404':
        description: Not Found
definitions:
  Product:
    type: object
    properties:
      count:
        type: integer
        format: int32
      desc:
        type: string
      id:
        type: integer
        format: int32
      name:
        type: string
```

上面的示例定义了两个 API，一个是获取 Product 的列表，一个是根据 id 获取 Product 的详情。

编辑完成后，可以得到如图 2-17 所示的文档界面。



图 2-17 Swagger 生成文档的示例

(3) 点击 file-download yaml 下载 YAML 文件。

(4) 点击 generate server 下载服务端, 点击 generate client 下载客户端, 可以分别生成相应语言的 SDK 工程。

## 2.7.6 通过 Spring Boot、Springfox、Swagger 实现 RESTful

上一节介绍了通过 Swagger Editor 实现 API 设计 (先写 YAML 文件, 然后生成服务端和客户端), 本节介绍另外一种直接写代码、通过注解自动生成相关文档的方法。

Spring Boot 已经家喻户晓, 而 Springfox 是什么呢? Marty Pitt 曾经编写了一个基于 Spring 的组件 swagger-springmvc, 用于将 Swagger 集成到 Spring MVC 中, Springfox 是从这个组件发展而来的。

下面我们通过一个简单的示例来说明一下。首先, 新建一个项目, 基于 Maven 构建, 在 pom 中引入相应的 JAR 包, 这里需要引入 Spring Boot 和 Springfox 的相关包。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger2</artifactId>
  <version>2.7.0</version>
</dependency>
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger-ui</artifactId>
  <version>2.7.0</version>
</dependency>
```

编写 Swagger 的配置类。

```
@Configuration
@EnableSwagger2
public class Swagger2 {
```



```
@Bean
public Docket createRestApi() {
    return new Docket(DocumentationType.SWAGGER_2)
        .apiInfo(apiInfo())
        .select()
        .apis(RequestHandlerSelectors.basePackage("com.cloudnative.rest"))
        .paths(PathSelectors.any())
        .build();
}

private ApiInfo apiInfo() {
    return new ApiInfoBuilder()
        .title("Swagger 构建 RESTful API")
        .description("")
        .termsOfServiceUrl("")
        .version("1.0")
        .build();
}
}
```

实现 dto 类。

```
public class Product {
    private int id;
    private String name;
    private int count;
    private String desc;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getCount() {
        return count;
    }

    public void setCount(int count) {
        this.count = count;
    }
}
```

```

    public String getDesc() {
        return desc;
    }

    public void setDesc(String desc) {
        this.desc = desc;
    }

    public int getId() {

        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    @Override
    public String toString() {
        return "Product{" +
            "id=" + id +
            ", name='" + name + '\'' +
            ", count=" + count +
            ", desc='" + desc + '\'' +
            '}';
    }
}

```

基于注解编写接口实现。

```

@RestController
@RequestMapping(value="/products")    //通过这里配置使下面的映射都在/products 下
public class ProductController {
    private List<Product> productList;
    //初始化
    public ProductController(){
        productList = new ArrayList<Product>();
        for (int i = 0; i < 10; i++) {
            Product product =new Product();
            product.setId(i);
            product.setCount(i+10);
            product.setName("watch"+i);
            product.setDesc("watch desc"+i);
            productList.add(product);
        }
    }
}

```



```

    }

    }

    @ApiOperation(value="获取产品列表", notes="获取产品列表")
    @RequestMapping(value="{", method= RequestMethod.GET)
    public List<Product> getProductList() {
        return productList;
    }

    @ApiOperation(value="获取产品详细信息", notes="根据 url 的 id 来获取产品详细信息")
    @ApiImplicitParam(name = "id", value = "产品 ID", required = true, dataType =
"Integer", paramType="path")
    @RequestMapping(value="/{id}", method=RequestMethod.GET)
    public Product getProduct(@PathVariable Integer id) {
        return productList.get(id);
    }
}

```

基于 Main 方法启动。

```

@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}

```

访问 <http://localhost:8080/swagger-ui.html>，自动生成的文档界面如图 2-18 所示。



图 2-18 生成的文档界面

基于图 2-18 的界面进行测试, 按 “Try it out!” 按钮执行, 如图 2-19 所示。

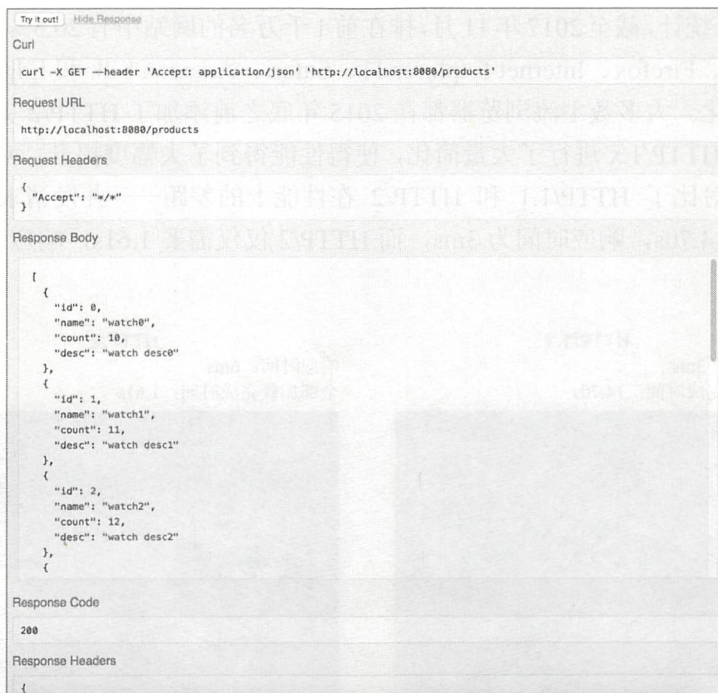


图 2-19 在生成的文档上进行测试

访问 <http://localhost:8080/v2/api-docs> 可以获取接口的 JSON 描述文件, 如图 2-20 所示。可以直接到 Swagger 官网把 JSON 描述文件转换为 YAML 文件。

```

{
  "swagger": "2.0",
  "info": {
    "version": "1.0",
    "title": "Swagger构建RESTful API",
    "host": "localhost:8080",
    "basePath": "/",
    "tags": [
      {
        "name": "product-controller",
        "description": "Product Controller"
      },
      {
        "name": "user-controller",
        "description": "User Controller"
      }
    ],
    "paths": {
      "/products": {
        "get": {
          "tags": [
            "product-controller"
          ],
          "summary": "获取产品列表",
          "description": "获取产品列表",
          "operationId": "getProductListUsingGET",
          "consumes": [
            "application/json"
          ],
          "produces": [
            "*/*"
          ],
          "responses": {
            "200": {
              "description": "OK",
              "schema": {
                "type": "array",
                "items": {
                  "$ref": "#/definitions/Product"
                }
              }
            },
            "401": {
              "description": "Unauthorized",
              "schema": {
                "type": "string"
              }
            },
            "404": {
              "description": "Not Found",
              "schema": {
                "type": "string"
              }
            }
          }
        },
        "/products/{id}": {
          "get": {
            "tags": [
              "product-controller"
            ],
            "summary": "获取产品详细信息",
            "description": "根据url的id来获取产品详细信息",
            "operationId": "getProductUsingGET",
            "consumes": [
              "application/json"
            ],
            "produces": [
              "*/*"
            ],
            "parameters": [
              {
                "name": "id",
                "in": "path",
                "description": "产品ID",
                "required": true,
                "type": "string"
              }
            ],
            "responses": {
              "200": {
                "description": "OK",
                "schema": {
                  "$ref": "#/definitions/Product"
                }
              },
              "401": {
                "description": "Unauthorized",
                "schema": {
                  "type": "string"
                }
              },
              "403": {
                "description": "Forbidden",
                "schema": {
                  "type": "string"
                }
              },
              "404": {
                "description": "Not Found",
                "schema": {
                  "type": "string"
                }
              }
            }
          },
          "post": {
            "tags": [
              "product-controller"
            ],
            "summary": "创建用户",
            "description": "根据url的id来创建用户",
            "operationId": "postUserUsingPOST",
            "consumes": [
              "application/json"
            ],
            "produces": [
              "*/*"
            ],
            "parameters": [
              {
                "name": "id",
                "in": "path",
                "description": "用户ID",
                "required": true,
                "type": "string"
              }
            ],
            "responses": {
              "200": {
                "description": "OK",
                "schema": {
                  "$ref": "#/definitions/User"
                }
              },
              "401": {
                "description": "Unauthorized",
                "schema": {
                  "type": "string"
                }
              },
              "403": {
                "description": "Forbidden",
                "schema": {
                  "type": "string"
                }
              },
              "404": {
                "description": "Not Found",
                "schema": {
                  "type": "string"
                }
              }
            }
          }
        }
      }
    }
  }
}

```

图 2-20 转换为 JSON 的结构



## 2.7.7 HTTP 协议的进化——HTTP/2

据 W3Techs 统计,截至 2017 年 11 月,排在前 1 千万名的网站中有 20.5%的支持 HTTP/2。Chrome、Opera、Firefox、Internet Explorer 11、Safari、Amazon Silk 和 Edge 浏览器都支持 HTTP/2 的标准化。大多数主流浏览器都在 2015 年底之前添加了 HTTP/2 支持。

HTTP/2 对 HTTP/1.x 进行了大量简化,使得性能得到了大幅度提升。Akamai 公司官网通过一个示例<sup>①</sup>对比了 HTTP/1.1 和 HTTP/2 在性能上的差距——并发请求 379 张图片,HTTP/1.1 需要 14.70s,响应时间为 3ms,而 HTTP/2 仅仅需要 1.61s,响应时间为 6ms,如图 2-21 所示。

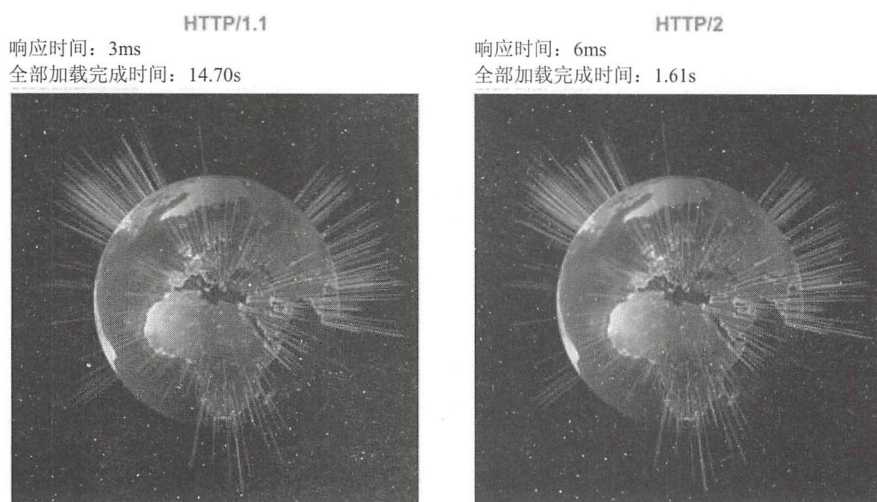


图 2-21 对比 HTTP/1.1 和 HTTP/2 的访问性能

我们都知道,在 HTTP/1.x 的协议中,浏览器在同一时间对同一域名下的请求数量是有限制的,这会导致大量并发请求阻塞,这个问题也被称为线端阻塞(head-of-line blocking)。同一域名下浏览器支持的连接数,如表 2-4 所示。HTTP/1.1 对不同浏览器连接数的限制不同,很多互联网公司为了解决这个问题,做了大量优化,包括建立多域名,通过 CDN 缓存大量静态资源等。

HTTP/2 是基于二进制协议的,与 HTTP/1.x 这样的文本协议相比,显然二进制协议性能更高。另外 HTTP/2 使用报头压缩,降低了网络开销。HTTP/2 将 HTTP 协议通信分解为二进制编码帧的交换,这些帧对应着特定数据流中的消息。所有这些消息都在一个 TCP 连

<sup>①</sup> 参见 <https://http2.akamai.com/demo>。

接内复用，这就是 HTTP/2 的多路复用机制（Multiplexing）。Benjamin 在 2015 年写的一篇文章中描述了一个简单的例子，如果只请求 3 个资源，从 Web 页面开始渲染到加载结束，HTTP/2 比 HTTP/1.1 节省不少时间，如图 2-22 所示。

表 2-4 同一域名下浏览器支持的连接数

浏 览 器	每个域名支持的最大并发连接
IE 9	6
IE 10	8
Firefox 4+	6
Opera 11+	6
Chrome 4+	6
Safari	4

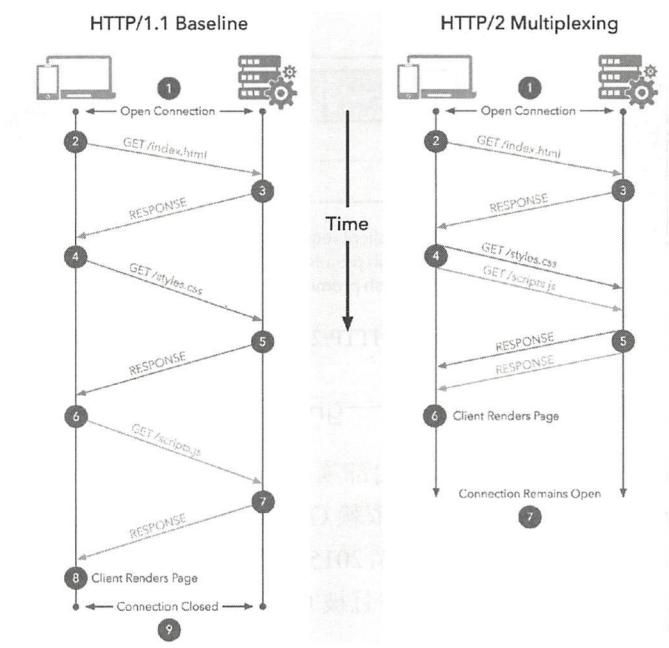


图 2-22 HTTP/1.1 和 HTTP/2 调用流程对比<sup>①</sup>

HTTP/2 完全兼容 HTTP/1.1 的语义，HTTP/2 和 HTTP/1.1 的大部分高级语法（例如方法、状态码、头字段和 URI）都是相同的。

<sup>①</sup> 来自 <https://cascadingmedia.com/insites/2015/03/http-2.html>。



HTTP/1.1 如果要实现长连接，需要设置 `Connection:keep-alive` 来控制长连接时间，超时就断开 TCP 连接。只有在客户端发起请求的时候，服务器端才会响应。所以就算一直给服务器发送心跳包以维持长连接，也不能用来推送，只有客户端不断发起请求给服务器端，服务器才会响应，这就是 pull 轮询的方式。

HTTP/2 引入服务端推送模式，即服务端向客户端发送数据，如图 2-23 所示。服务器可以对一个客户端请求发送多个响应，HTTP/2 打破了严格的请求-响应语义，支持一次请求-多次响应的形式。由于现如今的 Web 界面丰富多彩，加载的资源往往非常多，服务端实际上已经知道要推送什么内容，但 HTTP/1.x 的语义只支持客户端发起请求、服务端响应数据。HTTP/2 改变了这种模式，只需要客户端发送一次请求，服务端便把所有的资源都推送到客户端。服务器推送的缺点是，在客户端已经缓存了资源的情况下可能会有冗余。这个问题可以通过服务器提示（Server Hint）解决。

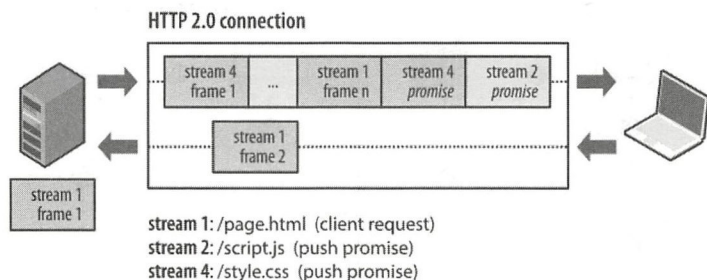


图 2-23 HTTP/2 推送模式<sup>①</sup>

## 2.7.8 HTTP/2 和 Protobuf 的组合——gRPC

gRPC 源于被称为 Stubby 的 Google 内部项目，Google 内部大量使用 Stubby 进行服务间通信。作为 gRPC 的前身，Stubby 大量依赖 Google 的其他基础服务，所以不太方便开放出来给社区使用。随着 HTTP/2 的逐步成熟，2015 年初 Google 开源了 gRPC 框架。截至 2017 年 12 月，gRPC 已经发布了 1.7.3 版本，并且被 CNCF（云原生计算基金会）所收录。gRPC 在 ETCD/Kubernetes 上得到了大量使用。

gRPC 是基于 HTTP/2 设计的，因此也继承了 HTTP/2 相应的诸多特性，这些特性使得其在移动设备上表现得更好，更节省空间、更省电。gRPC 目前提供的 C、Java 和 Go 语言版本分别是 `grpc`、`grpc-java`、`grpc-go`，其中 C 版本支持 C、C++、Node.js、Python、Ruby、Objective-C、PHP 和 C#。

<sup>①</sup> 来源 <https://hpbnc.co/http2/#server-push>。





说了这么多，gRPC 到底能够给我们提供哪些优势呢？

- gRPC 默认使用 Protobuf 进行序列化和反序列化，而 Protobuf 是已经被证明的高效的序列化方式，因此，gRPC 的序列化性能是可以得到保障的。
- gRPC 默认采用 HTTP/2 进行传输。HTTP/2 支持流（streaming），在批量发送数据的场景下使用流可以显著提升性能——服务端和客户端在接收数据的时候，可以不必等所有的消息全收到后才开始响应，而是在接收到第一条消息的时候就可以及时响应。例如，客户端向服务端发送了一千条 update 消息，服务端不必等到所有消息接收完毕才开始处理，而是一边接收一边处理。这显然比以前的类 HTTP 1.1 的方式提供的响应更快、性能更优。gRPC 的流可以分为三类：客户端流式发送、服务器流式返回，以及客户端/服务器同时流式处理，也就是单向流和双向流。在我写这本书的时候，Dubbo 3.0 正在酝酿中，其中一个显著的变化是新版本将以 streaming 为内核，而不再是 2.0 时代的 RPC，目的是去掉一切阻塞。
- 基于 HTTP/2 协议很容易实现负载均衡及流控的方案，可以利用 Header 做很多事情。

同时，gRPC 也不是完美的。相比于非 IDL 描述的 RPC（例如 Hessian、Kyro）方式，定义 proto 文件是一个比较麻烦的事情，而且需要额外安装客户端、插件等。另外 HTTP/2 相比于基于 TCP 的通信协议，性能上也有显著的差距。

下面通过一个简单的例子来理解一下 gRPC 的使用方式。假设我们要开发电商中的产品服务，通过 id 获取产品的信息，主要步骤及实现代码如下。

#### （1）定义 proto 文件。

```
syntax = "proto3"; //声明支持的版本是 proto3

option java_multiple_files = true; //以外部类模式生成
option java_package = "com.cloudnative.grpc"; //声明包名，可选
option java_outer_classname="ProductProtos"; //声明类名，可选

message ProductRequest{
    int32 id = 1;
}
message ProductResponse {
    int32 id = 1;
    string name = 2;
    string price = 3;
}

service ProductService{
    rpc GetProduct(ProductRequest) returns (ProductResponse);
```





```
}
```

(2) 生成相关类。可以采用 Protobuf 中介绍的方法，在命令行执行 `protoc` 生成相关代码。如果使用 Maven，则可以通过 Maven 插件实现。

```
<build>
  <extensions>
    <extension>
      <groupId>kr.motd.maven</groupId>
      <artifactId>os-maven-plugin</artifactId>
      <version>1.5.0.Final</version>
    </extension>
  </extensions>
  <plugins>
    <plugin>
      <groupId>org.xolstice.maven.plugins</groupId>
      <artifactId>protobuf-maven-plugin</artifactId>
      <version>0.5.0</version>
      <configuration>
        <protocArtifact>com.google.protobuf:3.5.1:exe:${os.detected.classifier}
</protocArtifact>
        <pluginId>grpc-java</pluginId>
        <pluginArtifact>io.grpc:protoc-gen-grpc-java:1.8.0:exe:${os.detected.
classifier}</pluginArtifact>
        <protocExecutable>/usr/local/bin/protoc</protocExecutable>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>compile</goal>
            <goal>compile-custom</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

在 `pom.xml` 中配置，并且执行 `mvn compile` 命令会在 `target/generated-sources` 中生成相关类，可以将相关类移到 `src/main/java` 目录下备用。

(3) 服务端实现代码。一是，实现 `ProductService`。

```
public class ProductService extends ProductServiceGrpc.ProductServiceImplBase{
    private static final Logger logger = Logger.getLogger (GRPCServer.class.getName());
```



```
@Override
public void getProduct(ProductRequest request, StreamObserver<ProductResponse>
responseObserver) {
    logger.info("接收到客户端的信息:"+request.getId());
    ProductResponse responded;
    if (111==request.getId()){
        responded=ProductResponse.newBuilder().setId(111).setName
("dddd").build();
    }else {
        responded=ProductResponse.newBuilder().setId(0).setName("---").build();
    }
    responseObserver.onNext(responded);
    responseObserver.onCompleted();
}
}
```

二是，实现 server 代码。

```
public class GRPCServer{
    private static final Logger logger = Logger.getLogger(GRPCServer.class.getName());

    private final int port;
    private final Server server;

    public GRPCServer(int port){
        this.port=port;
        this.server = ServerBuilder.forPort(port)
            .addService(new ProductService())
            .build();
    }
    /** Start serving requests. */
    public void start() throws IOException {
        this.server.start();
        logger.info("Server started, listening on " + port);
        Runtime.getRuntime().addShutdownHook(new Thread() {
            @Override
            public void run() {
                // Use stderr here since the logger may has been reset by its JVM shutdown
hook.

                logger.info("*** shutting down gRPC server since JVM is shutting down");
                GRPCServer.this.stop();
                logger.info("*** server shut down");
            }
        });
    }
}
```





```

    }

    /** Stop serving requests and shutdown resources. */
    public void stop() {
        if (server != null) {
            server.shutdown();
        }
    }

    /**
     * Await termination on the main thread since the grpc library uses daemon threads.
     */
    private void blockUntilShutdown() throws InterruptedException {
        if (server != null) {
            server.awaitTermination();
        }
    }

    /**
     * Main method. This comment makes the linter happy.
     */
    public static void main(String[] args) throws Exception {
        GRPCServer server = new GRPCServer(8888);
        server.start();
        server.blockUntilShutdown();
    }
}

```

#### (4) 客户端实现代码。

```

public class GRPCClient {
    private static final Logger logger = Logger.getLogger(GRPCServer.class.getName());
    public static void main(String[] args) {

        ManagedChannel channel = ManagedChannelBuilder.forAddress("localhost", 8888)
            .usePlaintext(true)
            .build();

        ProductServiceGrpc.ProductServiceBlockingStub
        blockStub=ProductServiceGrpc.newBlockingStub(channel);
        ProductResponse
        response=blockStub.getProduct(ProductRequest.newBuilder().setId(111).build());
        logger.info(response.getName());
    }
}

```





```
response=blockStub.getProduct(ProductRequest.newBuilder().setId(2).build());
logger.info(response.getName());

}
```

上面是一个简单的实现，关于流式 RPC 可以参考官方的例子<sup>①</sup>。

## 2.8 微服务框架

我们希望通过抽象出一个微服务框架，降低业务开发人员开发的复杂度，提升架构的下限。微服务框架有很多，综合起来看需要实现如下几个重要的功能。

### 服务治理

从单体架构演进到微服务架构时，服务数量可能会发生爆炸性增长，当服务达到成千上万个的时候，如何进行管理？架构师都不确定某个服务被谁调用了，升级需要通知谁。当服务提供者数量发生变化，上线下线发生时，如何动态通知消费方？是否需要重启消费方？这就需要微服务框架能够治理服务，通过注册中心实现服务的注册、发现。

### 容量规划

场景一：假设某服务最大承受 10 000TPS，如果超出是排队等待还是保证 10 000TPS 范围内请求正常，其他请求直接拒绝？

场景二：非核心服务流量洪峰导致核心服务受到影响，如何解决？

场景三：某服务有几百个消费者，如果其中一个消费者要求升级，满足一个临时活动，是否要求所有服务同步升级？

这就需要微服务框架能够设定最大容量，采用洪峰策略，隔离关键服务，控制版本，对服务分级，优雅降级。

### 高效通信

场景一：服务化后，调用链变长，原本的一次 RPC 通信可能变成几次甚至几十次，性能损失严重。

---

<sup>①</sup> 例子的地址是 <https://github.com/grpc/grpc-java/tree/master/examples>。



场景二：某些核心服务的一次请求大概需要调用几十个服务，如果按照串行方式，假设每个服务的响应时间为 50ms 返回，总耗时已经是秒级了该如何优化。

这就需要微服务框架实现高效的序列化、反序列化，支持并行、异步、非阻塞转换，支持多语言。

## 负载均衡

使用 F5 等硬件负载均衡非常容易导致性能瓶颈且价格昂贵，如果使用 Nginx、Apache、LVS 等软负载均衡很难扩展，很难实现个性化需求。这就需要微服务框架支持常用策略负载均衡、故障转移，支持自由的流量切换。

基于 RPC 框架，通过简单的封装，也可以实现大多数必需的功能。例如，某些互联网企业以 Thrift 作为 RPC 框架，在此之上实现 Failover、LoadBalance，以及 Retry 策略，通过 ZooKeeper、Etcd、Consul 实现服务注册和发现机制，同时将性能统计和降级开关等通用逻辑以 filter 的方式实现。

下面就带大家了解一下开源的微服务框架实现。

## 2.9 基于 Dubbo 框架实现微服务

Dubbo 是阿里早期服务化过程中开源的产品。2012 年，阿里巴巴在 GitHub 上开源了基于 Java 的分布式服务治理框架 Dubbo。当时服务化框架比较少，国内只有少数几个大型互联网公司采用自研的方式实现了服务化框架，所以 Dubbo 一开源就受到国内用户的追捧。虽然此后由于阿里内部的原因，Dubbo 一直处于冻结状态，但是由于 Dubbo 本身扩展性非常好，文档非常多，同类型开源框架选择不多，因此很多公司的服务化框架都是基于 Dubbo 改进的，或者参考 Dubbo 的架构思想，例如京东、当当的服务化框架。

Dubbo 中一共有四个角色，Dubbo 的角色说明如下。

- Provider，提供者。暴露服务的服务提供方（业务服务，通过 SDK 集成）。
- Consumer，消费者。调用远程服务的服务消费方（业务服务，通过 SDK 集成）。
- Registry，注册中心。负责服务注册与发现，有多种实现方式，包括 ZooKeeper、Redis 等。
- Monitor，监控。统计服务的调用次数和调用时间的监控中心，实现比较简单，不能用在生产环境中。

Dubbo 的调用流程<sup>①</sup>，如图 2-24 所示。

- (1) 提供者启动并向注册中心注册自己提供的服务，持久化到注册中心。
- (2) 消费者启动，查询注册中心。注册中心返回提供者地址列表给消费者，如有变更，注册中心将基于长连接推送变更数据给消费者。
- (3) 消费者基于负载均衡算法，从提供者地址列表中选择提供者进行调用。
- (4) 消费者和提供者在内存中累计调用次数和调用时间，定时发送统计数据到监控。

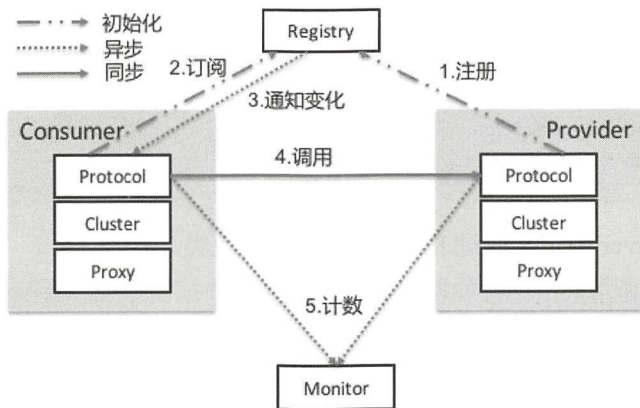


图 2-24 Dubbo 框架调用流程<sup>②</sup>

Dubbo 的设计思想一直被业界称赞，Dubbo 采用的是 JDK 标准的 SPI 扩展机制，微核心+插件式，平等对待第三方，扩展性极佳。Dubbo 还做了大量可靠性相关的设计，包括集群容错、并发控制、超时策略、线程池饱和策略等。

那么，如何通过 Dubbo 框架实现服务调用呢？以下描述了一个简单的步骤。

(1) 在开发环境中安装 JDK、Eclipse、Maven 等工具，如果协作开发需要安装 Maven 中央仓库和 GIT/SVN。

(2) 在开发环境中安装注册中心 ZooKeeper。

(3) 在开发环境中安装 dubbo-admin，通过图像界面管理 Dubbo。

(4) 基于 Maven 创建服务，假设要创建一个商品详情页，商品详情页依赖于价格服务和用户服务。因此在 Eclipse 中一共要创建三个 Project，分别是 product-detail、price、user。以 user 为例，总共分为三个 module，如图 2-25 所示。其中，user-api、user-dto 作为 client（合成一个 module 也可以），通过 Maven 提交到公共的库中供消费者使用；user-server 是

<sup>①</sup> 参考 Dubbo 官网 <http://dubbo.apache.org/#!/docs/user/preface/architecture.md?lang=en-us>。

<sup>②</sup> 图片来自 <http://dubbo.apache.org/#!/docs/user/preface/architecture.md?lang=en-us>。





user 的提供者服务。



图 2-25 项目结构

(5) 分别在三个 Project 里引入相关 JAR 包。

(6) 在 user-api 中定义通过 ID 获取 user 的接口。

```
public interface UserService {
    public User getUserById(Long id);
}
```

(7) 在 user-server 中定义通过 ID 获取 user 的实现类。

```
public User getUserById(Long id) {
    User user = new User();
    user.setId(1L);
    user.setName("www");
    return user;
}
```

(8) 在 user 里配置启动函数。

```
public static void main(String[] args) {
    ApplicationContext applicationContext = new ClassPathXmlApplicationContext(
("spring.xml");
    System.out.println(new SimpleDateFormat("[yyyy-MM-dd HH:mm:ss]").format(new Date()))
+ " Dubbo service server started!");
    synchronized (Main.class) {
        try {
            Main.class.wait();
        } catch (Throwable e) {
        }
    }
}
```

(9) 在 product-detail 中创建接口 ProductDetailService。

```
public interface ProductDetailService {
    public Map getProductDetail( Long id);
}
```

实现这个接口。

```
@Path("detail")
public class ProductDetailServiceImpl implements ProductDetailService{
    @Resource
    private PriceService priceService;
    @Resource
    private UserService userService;
    @GET
    @Path("{id : \\d+}")
    @Produces({MediaType.APPLICATION_JSON})
    public Map<String, Object> getProductDetail(@PathParam("id") Long id) {
        Map<String, Object> result=new HashMap<String, Object>();
        Price price = priceService.getPriceByProductId(1L);
        result.put("price", price);
        result.put("user", userService.getUserById(1L));
        return result;
    }
}
```

(10) 在 product-detail 中创建配置文件，如图 2-26 所示。



图 2-26 配置文件目录

(11) 在 spring.xml 中做如下配置。

```
<!-- 属性文件读入 -->
<bean class="org.springframework.beans.factory.config.PropertyPlaceholder
Configurer">
    <property name="locations">
        <list>
            <value>classpath*:dubbo.properties</value>
        </list>
    </property>
    <property name="fileEncoding">
        <value>utf-8</value>
    </property>
</bean>
<import resource="classpath*:spring-dubbo.xml" />
```

(12) 在 `spring-dubbo.xml` 中做如下配置。

```
<!-- 提供方应用信息，用于计算依赖关系 -->
<dubbo:application name="price-service-provider" />
<!-- 使用 zookeeper 注册中心暴露服务地址 -->
<dubbo:registry protocol="zookeeper" address="${zookeeper.url}"
    timeout="${registry.timeout}" />
<!-- 暴露用 dubbo 协议服务端口号 -->
<dubbo:protocol name="dubbo" port="${dubbo.port}"
    threads="${dubbo.threads}" />
<!-- 我用的是 dubbox，最新 dubbo 已经合并了 dubbox 提供 rest 功能 -->
<dubbo:protocol name="rest" port="${rest.port}" server="tomcat"
    threads="${rest.threads}" />
<!-- 声明需要暴露的服务接口 -->
<dubbo:service interface="com.huawei.product.service.ProductDetailService"
    ref="productService" protocol="rest" />
<dubbo:annotation package="*" />
<bean id="productService" class="com.huawei.server.service.ProductDetail
ServiceImpl"></bean>
<!-- 超时时间 5s，dubbo 默认采用的是单一长连接，可以修改连接数 10-->
<dubbo:reference id="priceService" interface="com.huawei.price.api.PriceService"
    timeout="5000" connections="10" />
<dubbo:reference id="userService"
    interface="com.huawei.user.service.UserService" timeout="20000" connections="10"
/>
```

(13) `price` 和 `user` 类似，在 `product` 创建 `Main` 函数启动，整个流程就都跑起来了。

另外，Dubbo 也支持注解配置。

由于 Dubbo 是阿里早期服务化过程中开源的产品，严格意义上来讲，Dubbo 对于目前的微服务架构理念并不是完全的适配。可喜的是 Dubbo 又开始活跃了，计划贡献给 Apache，而 Dubbo 现在定位是 RPC 框架，而不是像 Spring Cloud 那种全集。

## 2.10 基于 Spring Cloud 框架实现微服务

有一个对 Spring Cloud 非常形象的称呼是微服务“全家桶”，非常贴切。Spring Cloud 依托于 Spring Boot 提供了很多工具、组件、服务给开发者，以快速、简单地实现微服务架构。Spring Cloud 包含了很多子项目，如下所示。

- Spring Cloud Config：配置管理工具。
- Spring Cloud Netflix：整合了 Netflix 的微服务相关套件。
- Spring Cloud Bus：消息总线，提供消息机制进行集群间通信。





- Spring Cloud Cluster: 选举算法服务, 包括 ZooKeeper、Redis、Hazelcast、Consul。
- Spring Cloud Consul: 基于 Consul 的服务注册中心。
- Spring Cloud ZooKeeper: 基于 ZooKeeper 的服务注册中心。
- Spring Cloud Security: 安全工具包。
- Spring Cloud Sleuth: 适配 Zipkin、HTrace 和 ELK 的服务调用链跟踪。
- Spring Cloud Data Flow: 一个基于 Cloud Native 的可编程服务, 用于数据处理。
- Spring Cloud Stream: 适配 Kafka、RabbitMQ 轻量级消息中间件框架。
- Spring Cloud Stream App Starters: Spring Cloud Stream 基于 Spring Boot 启动的独立进程。
- Spring Cloud Task: 短生命周期的微服务框架, 也就是定时任务框架。
- Spring Cloud Task App Starters: Spring Cloud Task 基于 Spring Boot 启动的独立进程。
- Spring Cloud Starters: 基础启动组件, 实现 Spring Boot 风格的依赖管理。
- Spring Cloud Netflix: 封装了许多 Netflix OSS 微服务框架的组件、服务。下面通过一个实例展示如何通过 Spring Cloud Netflix 开发一个微服务。

在 Netflix 中, 所有服务都是一样的, 包括 Eureka, 没有生产者和消费者的区别。为了便于大家理解, 将生产者和消费者区分开来, 开发以下三个服务。

- 注册中心, 基于 Eureka 实现, 可以通过界面看到自身的相关信息, 也可以看到服务列表及状态信息。
- 生产者, 需要满足 Eureka 的通信机制, 注册到 Eureka 上。
- 消费者, 首先注册到 Eureka 上, 请求可用的提供者服务列表, 然后调用提供者返回结果。

首先, 启动服务注册中心, Netflix 的注册中心是 Eureka, 即启动 Eureka。

(1) 新建 Project, 引入相关依赖包。

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.5.2.RELEASE</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
<modelVersion>4.0.0</modelVersion>
<artifactId>spring-cloud-demo</artifactId>
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <java.version>1.8</java.version>
</properties>
```



```

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Camden.SR6</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka-server</artifactId>
  </dependency>
</dependencies>

```

(2) 定义配置文件，Spring Boot 有两种配置方式，一种是基于 properties 的，一种是基于 YAML 的，这里选择基于 YAML 新建 application.yml。

```

server:
  port: 10001 #定义端口
eureka:
  client:
    registerWithEureka: false
    fetchRegistry: false
    serviceUrl:
      defaultZone: http://localhost:10001/eureka

```

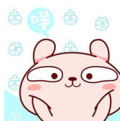
(3) 定义 Main 函数，启动注册服务。

```

@SpringBootApplication
@EnableEurekaServer
public class EurekaServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(EurekaServiceApplication.class, args);
    }
}

```





(4) 检查服务启动情况, 此时可以访问 <http://localhost:10001>。Eureka 的注册中心界面, 如图 2-27 所示。

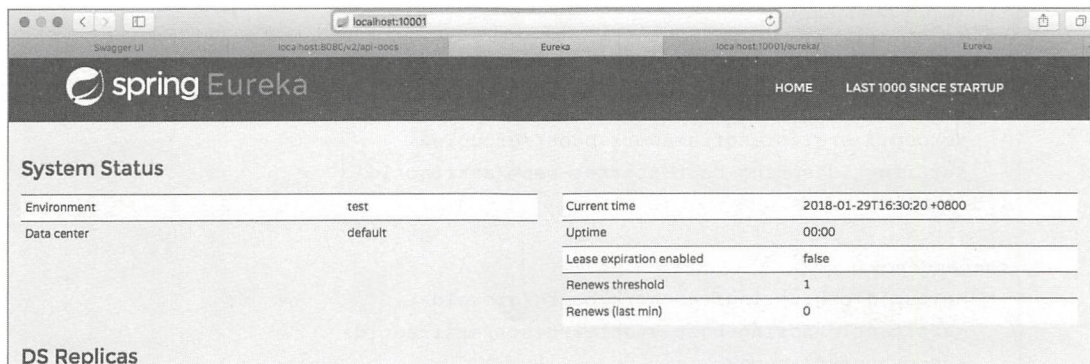


图 2-27 注册中心界面

以上, 我们就启动了一个注册中心, 用于服务的注册发现。  
其次, 实现服务生产者。

(1) 引入依赖包。

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.5.2.RELEASE</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
<modelVersion>4.0.0</modelVersion>

<artifactId>eureka-provider</artifactId>

<properties>
  <!-- 设置字符编码及 Java 版本 -->
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <java.version>1.8</java.version>
</properties>
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Camden.SR6</version>
      <type>pom</type>
```





```
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
  </dependency>
</dependencies>
```

## (2) 定义配置文件 application.yml。

```
server:
  port: 10002 #定义端口
spring:
  application:
    name: provider
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:10001/eureka
  instance:
    metadataMap:
      instanceId: ${spring.application.name}:${random.value}
```

## (3) 实现 Controller。

```
@RestController
public class PriceController {

    @RequestMapping(value="/price/{id}",method = RequestMethod.GET)
    public String getPriceByID(@PathVariable Long id){
        System.out.println("id = [" + id + ""]);
    }
}
```



```

        return "{id:"+id+"}";
    }
}

```

(4) 定义 Main 函数，启动生产者。

```

@EnableDiscoveryClient
@SpringBootApplication
public class ProviderApplication {
    @Autowired
    private DiscoveryClient discoveryClient;

    public String serviceUrl() {
        List<ServiceInstance> list = discoveryClient.getInstances("PROVIDER");
        if (list != null && list.size() > 0 ) {
            return String.valueOf(list.get(0).getUri());
        }
        return null;
    }

    public static void main(String[] args) {
        SpringApplication.run(ProviderApplication.class, args);
    }
}

```

注意，serviceUrl 方法中 getInstances 的名字是 PROVIDER，它是 YAML 文件配置的，从 Eureka 管理界面中也能看到。如果名字不对，那么后面消费者会找不到提供者。

(5) 检查服务启动情况，status 显示为 up，表示已上线，如图 2-28 所示。

#### Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
PROVIDER	n/a (1)	(1)	UP (1) - 172.20.10.3:provider:10002

图 2-28 服务状态

(6) 通过 RESTful 接口请求地址 <http://127.0.0.1:10002/price/1>，页面返回 {id:1}，表示成功。

最后，实现服务消费者。

(1) 引入依赖包。这里要用到 Ribbon，Ribbon 是实现客户端负载均衡的组件。

```

<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>1.5.2.RELEASE</version>

```





```
<relativePath/> <!-- lookup parent from repository -->
</parent>
<modelVersion>4.0.0</modelVersion>

<artifactId>eureka-consumer</artifactId>
<properties>
  <!--设置字符编码及 Java 版本-->
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <java.version>1.8</java.version>
</properties>
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Camden.SR6</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>

  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-ribbon</artifactId>
  </dependency>
</dependencies>
```





## (2) 定义配置文件 application.yml。

```
server:
  port: 10003 #定义端口
spring:
  application:
    name: consumer
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:10001/eureka

  instance:
    metadataMap:
      instanceId: ${spring.application.name}:${random.value}
```

通过 `instance` 可以配置应用在配置中心显示的名字，因为加入了随机数，所以很容易区分开。

## (3) 实现 Controller。

```
@RestController
public class ProductDetailController {
    @Autowired
    RestTemplate restTemplate;

    @RequestMapping(value = "/detail/{id}", method = RequestMethod.GET)
    public String getProductDetail(@PathVariable long id){
        System.out.println("id = [" + id + "]");
        return
        restTemplate.getForEntity("http://PROVIDER/price/"+id, String.class).getBody();
    }
}
```

注意这里的地址是 `PROVIDER`，不是一个具体的 IP 和端口，这样才能负载均衡，如果直接写具体的 IP 和端口可以定位到一个唯一的服务实例。

## (4) 定义 Main 函数，启动消费者。

```
@EnableDiscoveryClient
@SpringBootApplication
public class ConsumerApplication {

    @Bean
    @LoadBalanced
```



```
RestTemplate restTemplate() {
    return new RestTemplate();
}

public static void main(String[] args) {
    SpringApplication.run(ConsumerApplication.class, args);
}
}
```

(5) 检查服务启动情况，生产者和消费者都显示 up，表示已成功注册，如图 2-29 所示。

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
CONSUMER	n/a (1)	(1)	UP (1) - 172.20.10.3:consumer:10003
PROVIDER	n/a (1)	(1)	UP (1) - 172.20.10.3:provider:10002

图 2-29 服务状态

(6) 通过 RESTful 接口请求地址 `http://127.0.0.1:10003/detail/2`，页面返回 `{id:2}`，表示成功。当然，Eureka 也可以是集群，也可以分布到不同的地域，如图 2-30 所示。

如果服务同时注册到多个 Eureka，配置中 `default-zone` 通过逗号隔开多个注册中心地址即可。

```
eureka:
  client:
    service-url:
      default-zone:http://registry1:10001/eureka,http://registry2:10002/eureka
```

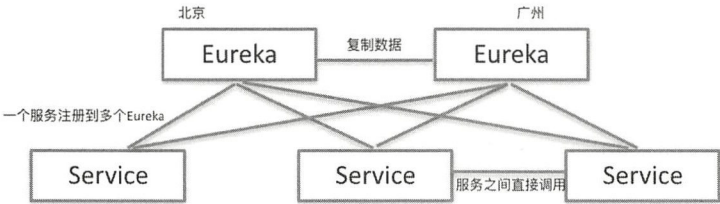


图 2-30 多注册中心

在 Eureka 中，因为我们要启动两个实例互相注册，所以需要不同的域名，如下所示，分别配置 `registry1` 和 `registry2` 两个域名。在一台机器上启动两个服务，可以用 `profile` 区分。

```
---
spring:
  profiles: registry1
```





```
eureka:
  instance:
    hostname: registry1
  client:
    serviceUrl:
      defaultZone: http://registry2/eureka/

---
spring:
  profiles: registry2
eureka:
  instance:
    hostname: registry2
  client:
    serviceUrl:
      defaultZone: http://registry1/eureka/
```

## 2.11 服务发现场景下的 ZooKeeper 与 Etcd

早期,在服务发现领域, ZooKeeper 一直以来都是不二之选,近几年,由于 ZooKeeper 发展速度放缓, Docker 的崛起引爆了 Etcd 这个服务发现组件,下面就对这两个组件做一下分析。

Etcd 是一个高可用的键值存储系统,主要用于共享配置和服务发现。Etcd 是由 CoreOS 开发并维护的,灵感来自 ZooKeeper 和 Doozer,它使用 Go 语言编写,并通过 Raft 一致性算法处理日志复制以保证强一致性。

ZooKeeper 是 Apache 软件基金会的一个软件项目,为大型分布式计算提供开源的分布式配置服务、同步服务和命名注册。ZooKeeper 曾经是 Hadoop 的一个子项目,但现在是一个独立的顶级项目。ZooKeeper 的架构通过冗余服务实现高可用性。

ZooKeeper 真的是高可用吗?不,ZooKeeper 实际上是一个 CP<sup>①</sup>的系统,通过 Paxos(zab)来保证一致性,也就是参议员投票机制。假设在 3 个数据中心分别部署了 3 个 ZooKeeper 节点(为了容灾),当发生网络分区的时候,也就是其中一个节点和其他两个断网了,已经连接到这个节点的应用将无法向这个节点读写数据。作为一个分布协调系统,做到这点是必须的。实际上在客户端缓存数据是一个折中的办法,大多服务化框架也是这么做的,但是在服务发现场景下,正在启动的服务就非常不幸了。而这种场景并不少见,特别是当某

---

① 根据 CAP 理论,三者只能选择两者,CP 表示优先满足一致性和分区容忍性。





个数据中心出现大面积故障时。而 Etcd 虽然是不可写的，但是可以读取数据。在服务发现场景下，能读取到不一致的数据要好于什么都读不到。

ZooKeeper 的故障恢复时间是惊人的。一旦网络发生抖动，触发选主流程，意味着整个集群将会是不可用状态，而选主的时间通常在 30~120s 之间，在这段时间，应用会不断报出异常，如果分析生产环境上的日志，就会发现这个问题。而 Etcd 依赖 Raft 算法，选主流程要快得多。Raft 竞选过程中，如果发生票数相同的情况，那么竞选节点（Candidate）Time Out 时间采用随机值，有效降低了冲突的概率。

Etcd 并不是强一致的。假设有 5 个节点，发生网络分区后，其中 Leader 节点被分到了少数区域，此时多数节点又选出了一个新的 Leader。在这种情况下，存在两个 Leader，新的 Leader 在下次 heartbeat timeout 时向所有的节点发送一次 heartbeat，Leader 在收到步进数更高的 Leader heartbeat 时放弃 Leader 地位并切换到 Follower 状态。值得注意的是，步进数落后的 Leader 将丢弃前面的数据，和 Leader 保持一致。

从系统容灾的角度看，实际上对两个系统来说都能做到，只是需要适当延长超时时间和心跳间隔。Etcd 的官网给出了具体的方案<sup>①</sup>。不过对于 ZooKeeper 来说，由于上文中提到的一些观点，可用性、性能受到挑战，很多公司采用在单个区域形成集群，跨区域增加 observer 的方式来提升读性能。例如在北京地区的三机房部署一套集群，在杭州增加 observer 的方式，优先访问本地节点。这样需要在客户端设置优先策略，否则客户端访问 server list 将采用随机后轮询的机制。

再回到服务发现这个应用场景，在此场景下，Etcd 是一个不错的选择，一个理由参见上文。另一个理由是 Apache 的项目更新速度太慢，很多问题长时间得不到解决，导致很多用户自行修改源码，或者直接放弃。要知道，在很多公司，凭几个技术人员的几句话完全可以放弃一个存在个别问题的成熟开源项目，转为自己开发。也是因为 Etcd 的更新速度比较快，架构和接口容易变更，所以会有一定的成本。

## 2.12 微服务部署策略

在单体架构或者 SOA 架构中，由于服务数量比较少，通常采用运行多个实例扩展的策略，管理、运维都比较简单，可以在一个物理机或者虚拟机上部署一个服务，也可以部署多个，并没有一个明确的方案。但是演进到微服务架构之后，由于服务数量众多，开发人员开始思考部署策略问题。目前业界多种部署策略并存，下面将会分析一下各种策

---

① 参见 <https://coreos.com/etcd/docs/latest/tuning.html>。

略的优缺点。

2.12.1 服务独享数据库

服务独享数据库是指每个服务独享一个数据库，接口是对外提供数据共享的唯一方式，如图 2-31 所示。在服务独享数据库中，服务 2 如果需要查询服务 1 的数据，则必须通过接口进行访问。服务共享数据库是指多个服务的数据统一放到一个数据库中，服务之间可以直接通过数据库共享数据，如图 2-32 所示。

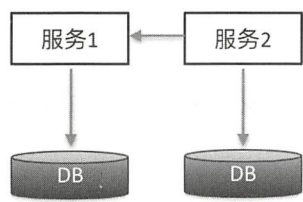


图 2-31 独享数据库

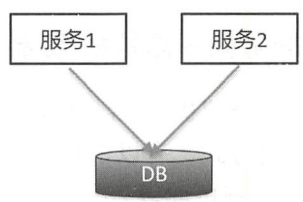


图 2-32 共享数据库

微服务架构推荐采用每个服务独享数据库的方式，这样做的原因有很多。

从常规来看，根据架构的成本、复杂度，一般的架构发展到微服务阶段，说明已经具备了一定的规模，服务拆分的原因很可能是数据库的扩展性受到了制约。在分表之前，应该先进行分库，服务拆分是这个阶段的必备“良药”，所以这个阶段采用服务独享数据库是比较合理的做法。

但是这并不意味着服务绝对不能共享数据库，以下两种情况可能会存在。

- 单体架构迁移到微服务架构的过渡阶段可以通过共享数据库的方式，利用存储过程、触发器等迁移数据。
- 在小规模部署的过程中存在服务共享数据库在交付型的场景中，往往要求一套架构适配多种场景，如果面临的是小规模部署，最好能够通过合并服务、共享数据库等方式降低资源占用。

表 2-5 对服务独享数据库和服务共享数据库进行了详细的对比。

表 2-5 服务独享数据库和服务共享数据库对比

指标	服务独享数据库	服务共享数据库	说 明
隔离性	高	低	服务独享数据库隔离性更好，例如某个业务出现故障不会影响其他业务。另外，当生产环境出现问题进行调试的时候，针对数据库的调试，故障也更容易排除
资源利用率	低	高	切分的粒度越小，整体的资源利用率越低

续表

指标	服务独享数据库	服务共享数据库	说 明
耦合度	低	高	服务共享数据库的策略只能通过规范的方式规定服务 1 不能调用服务 2 的表，约束力不足，会导致耦合度比较高，一旦绕过接口进行调用，将来服务 2 相关的演讲都需要通知服务 1，服务 1 会受到影响
扩展性	高	低	服务独享数据库可以针对业务的特色、场景进行扩展，例如如果通讯录访问量较高，可以只针对通讯录进行读写分离，也可以将通讯录存储在 Cassandra 中

### 2.12.2 服务独享虚拟机/容器

服务共享虚拟机/容器是指在一个虚拟机或容器中部署多个服务。服务共享虚拟机/容器是传统的部署方式，有以下几种情况。

- 一个物理机部署多个服务——进程级隔离。
- 一个虚拟机部署多个服务——进程级隔离。
- 一个系统容器（Docker 等）部署多个服务——进程级隔离。
- 一个应用容器（包括 Apache、Tomcat 等）部署多个服务——非进程级隔离。

这些部署方式更节省资源，适合规模较小的业务场景，缺点就是隔离性差，当服务中出现了死循环会导致 CPU 被占满。虽然服务之间属于进程级隔离，但是如果共享服务器，也不能隔离，服务同样会受到影响，并且部署比较复杂，开发和运维需要详细沟通。

服务独享虚拟机/容器是指在一个虚拟机或容器中只部署一个服务。

微服务架构一个比较重要的优点是隔离性，而服务独享虚拟机/容器把隔离级别提升到了虚拟机或者容器。另外它可以将服务打包成一个镜像，降低了安装部署的复杂度。当服务以天为单位发布时，这个优点就变得很重要。

服务独享虚拟机/容器的缺点是资源利用率不高。

## 2.13 为什么总觉得微服务架构很别扭

很多系统迁移到微服务架构之后，并没有明显感觉到微服务架构带来的优势，反而觉得带来了更高的复杂度，以下是微服务架构没能发挥出固有优势的原因。

### 用传统方式构建微服务

微服务架构和传统的架构方式思路完全不一样。例如传统方式实现高可用，更相信流



程，更相信 KPI<sup>①</sup>对人的影响，因此流程需要让更多的人去测试，制定更严格的发布流程。而微服务架构强调的是自动化发布、灰度发布、Design For Failure、自动化测试、故障隔离、自愈。很多失败的案例以传统的方式去构建微服务架构，一切都没有转变，只是把服务拆开，根本无法享受微服务架构带来的便利，反而因此遇到了更多麻烦。最明显的是对开发人员的影响，他们质疑微服务架构是否适合自己的业务场景，而一个充满质疑的团队是不可能具备强大战斗力的。

## 组织结构不变

如果要充分发挥微服务架构的优势，组织结构必须发生转变，构建和微服务配套的小团队，并且让他们拥有绝对的自主权。实际上，相当一部分实施微服务架构的团队都没有做到这一点，因为组织结构总是涉及利益。比较典型的问题是，小团队不需要团队以外的任何人来批准是否上线，架构如何演进，使用什么数据库。如果小团队没有权利，任何变动都要等待高层进行决策，就会形成决策瓶颈点，导致效率低下，这违背了微服务架构的初衷，团队成员也会因此失去主动性。

## 习惯于领导安排工作

传统的研发模式严重依赖流程，原因是没有人愿意承担责任，所有人都把责任推到流程上。微服务架构和敏捷开发流程是天作之合，传统研发模式需要领导批准，然后由团队负责人直接分解任务、定工时、安排任务负责人，而按照敏捷开发流程，开发计划应该是团队决定的，任务自主认领。精英化团队绝不仅仅是团队人数更少，人员能力更强这么简单，这只是表面上的，更重要的是责任感和主观能动性以及信任！

某公司 CEO 曾经说过这样一件事情，员工把请假流程提给“我”，让“我”审批，但是“我”根本没有审批的必要，因为你随便说一个理由“我”都不可能拒绝，就算拒绝了，下回会来一个更难以拒绝的理由。后来公司所有的请假都不需要审批了，只要在群里发一条消息，让相关同事知道就可以了。

## 纠结如何拆分服务

不理解微服务架构的人，通常从字面理解，他们认为微服务架构的重中之重就是服务拆分。

到底拆分多细？与其浪费更多的时间思考这个问题，不如先拆出几个服务运行一下，

---

<sup>①</sup> 一种绩效考核方式，Key Performance Indicator 的缩写。

感受一下。架构是一个持续的过程，有时候很难从技术角度完全解释清楚。

另一个错误是一次性拆分，不能改变。由于技术人员对业务领域知识的理解不断加深，业务逻辑有可能运行一段时间也会转变，这时候改变是不可避免的。重新合并、划分，是一个正常的演变过程。架构是动态的，不是静态的。

## 以大规模拆分服务开始

微服务架构需要一个适应过程，持续拆分效果更佳。如果从大规模拆分服务开始，需要具备三个条件，否则可能会遇到相当多的麻烦。

- 团队有微服务架构经验。
- 已具备微服务架构的先决条件，包括自动化的研发环境、全面的健康检查、必要的公共服务及框架，以及敏捷基础设施。
- 业务目标非常明确，已经可以预期未来的规模，业务几乎无变化。

## 高估架构的移植性

架构是一门艺术，不是随随便便复制一下就可以的，Google、Amazon、Facebook 的架构问世已经很长时间了，研发人员跳槽这么频繁，但是没有哪个公司能模仿好。MySQL 开源了这么多年，放到不同人的手里，结果完全不一样。实际上实施微服务架构在一个业务场景中的优势在另一个场景中很可能会变成一种劣势。如果你仔细研究就会发现，大多数公司实施的微服务架构就跟各个公司的管理制度一样各不相同。很多公司为了显示自己的架构有多厉害而实施微服务架构，最终只会害了团队，因为只把精力放在微服务架构上，可能就减少了对业务实现、用户体验的关注。

## 从来没有做过微服务架构的人领导你完成迁移

如果转型的团队由没有经验的人来领导，那么结果就是只关注表面，拆了多少服务、服务粒度、服务注册发现、负载均衡、调用链分析，而隐藏的各种性能问题、扩展性问题、可用性问题都没有得到足够的关注。如果只是从几个服务拆分开始积累经验还好，一旦大规模拆分，就会让整个团队都质疑微服务架构的意义。架构是需要实践的，不要以为看几篇文章就得到了架构的真谛，细节会“杀死”团队。

# 3

## 第3章 敏捷基础设施及公共 基础服务

未来软件研发的角色会越来越少，测试没有了，运维没有了，他们的工作由谁来做？为什么会有这样的预测？敏捷基础设施及公共基础服务将带给我们部分答案。

第2章介绍了微服务架构，向微服务架构转型的过程中会遇到各种各样的问题，敏捷基础设施及公共基础服务是微服务架构的有力支撑，可以说，敏捷基础设施及公共基础服务是微服务架构成败的关键因素之一，它能够简化业务开发，提升架构能力的下限。简单来说，就是一个普通的业务开发团队通过敏捷基础设施及公共基础服务可以快速开发出一个不平凡的产品，下面逐步讲解如何通过敏捷基础设施及公共基础服务提升团队能力的下限。

### 3.1 传统基础设施面临的挑战

任何新事物的创造，都是为了解决固有的问题，敏捷基础设施的诞生，同样是为了解决传统基础设施面临的挑战。以下是传统基础设施经常遇到的挑战。

- 资源利用率低，指多个未完全利用的服务器占据多于其工作量对应的空间，消耗较多资源的情况。对于大量的服务器，缺乏弹性及统一有效的管理会导致整体资源利



用率不高。业务上线时如果增加服务器需要层层审批，则会造成效率低下。云带给开发人员最直观的感受就是不需要每年做大量的资源规划了。

- 服务器数量呈爆炸性增长。由于互联网的发展，微服务架构的诞生，单体架构下的服务器被拆分为容器或虚拟机，导致需要管理的节点数呈爆炸性增长，传统的方式难以管理。
- 没有标准化。如果生产环境的服务器配置是人工管理的，就会存在一种场景：突然发生了某个事故，导致运维人员在服务器上做了很多操作，这样服务器之间就存在某种差异，最终导致程序只能在某个环境的某个服务器中才能正常运行。传统基础设施允许运维人员做各种特殊配置，导致所有服务器都不一样，就像天空飘下来的雪花一样，各不相同。
- 脆弱的基础设施。传统运维方式由于缺乏自动化手段，配置服务器较慢，一旦发现问题，就需要花大量的时间。另外，人工管理更容易出错。因此，传统运维方式给我们的直观感受就是基础设施非常脆弱。

## 3.2 什么是敏捷基础设施

在微服务架构爆发式增长的今天，大多数人看到了微服务架构带来的优势，重视服务拆分，而基础设施往往被忽略，但是它的重要程度一点都不比微服务架构低。实际上，部署、运维是非常浪费时间及精力的，并且非常重要，不能出错。如前所述，Cloud Native 的基石是微服务架构、敏捷基础设施及公共基础服务。

那敏捷基础设施到底能给我们提供什么价值呢？我们先说说基础设施运维的阶段。

第一阶段，“人肉”阶段。全部“人肉”，物理机安装软件，有专门的运维团队负责部署：A 物理机是给订单用的，B 物理机是给登录用的，绝对不能互相干扰。这个阶段常常因为程序员写错命令导致故障。标准化通过规范约束，效果甚微，效率十分低下。

第二阶段，脚本阶段。内部制定规范，要求必须严格执行。通过部分脚本实现部署、启动、停止。部署是半自动化方式，还是要通过运维人员操作、配置，仍然需要写命令。使用虚拟机隔离，虚拟机数量很多，运维人员在窗口中来回切换，可能会看错了窗口，执行了错误的命令。申请机器需要提前，每年都要做服务器需求计划，中间加机器非常麻烦。

第三阶段，工具阶段。少数运维人员通过私有云管理虚拟机，通过 CI 工具实现持续部署。运维人员通过虚拟机镜像来封装常用依赖环境。但是开发环境和测试环境、生产环境差距很大，可能开发人员本地测试通过，测试人员却发现了问题，或者测试人员在测试环境测试通过，上线后发现有问题。

第四阶段，敏捷基础设施阶段。无须运维人员，全部自动化，通过容器封装环境，开发人员可以直接将所有软件和依赖直接封装到容器中，打包成镜像，生产环境直接部署镜像，通过容器实现开发、测试、生产环境的一致。通过容器调度平台管理容器，资源利用率更高，通过配置文件描述环境，例如部署 8 台 Nginx，端口是什么，镜像用哪个，日志放在什么地方，配置文件用哪个，部署在什么地方等，都可以直接描述出来。注意，这个描述文件以前是运维干的，现在开发就能搞定。

敏捷基础设施实际上并不是一个全新的术语，它是指使用脚本或文件配置计算基础设施环境，我们可以认为这些脚本和文件就是代码，这些代码可以放入版本库，替换所有人有的命令行操作，使基础设施具有不变性，只要发布就不允许更改，若要更改需要重新发布。

敏捷基础设施也可称为基础设施即代码（Infrastructure as Code）或者可编程基础设施（Programmable Infrastructure）。基础设施即代码可以将基础设施配置完全当作编写代码来进行，这样做是为了提高效率，因为在整个研发环节，人与人特别是不同的角色之间的沟通是非常浪费时间的，参与沟通的人越少效率就越高。而基础设施即代码可以让一个全栈工程师在没有运维人员参与的情况下，申请生产环境的资源，自动化配置，部署应用。

这样做的好处是流程简单了，速度比以前快了，而且摒弃了烦琐的人工操作，一切都可以通过工具完成，更容易形成标准化，比人工操作更可靠。

基础设施即代码和传统的配置管理有一个非常大的区别，那就是整个过程由开发人员负责，无须运维人员参与，传统的配置管理由运维人员操作，效率低。这种做法意味着业务服务与基础设施的关系更加紧密了，开发人员不仅可以写业务服务的代码，还可以定义运行业务服务的基础设施。

基础设施即代码要想普及，还面临一些挑战，特别是要调整研发流程，改变角色的职责。很多失败的案例都是由于利益之争，国内某旅游互联网公司进行的基础设施即代码比较成功，他们的策略是让有代码能力的运维人员去开发云平台和工具，让没有代码能力的运维人员做基础运维，也就是传说中的“搬机器”，所有的开发流程都是由全栈工程师主导，取得了比较好的效果。

### 3.3 基于容器的敏捷基础设施

敏捷基础设施的目标如下。

- 标准化。所有的基础设施最好都是标准的，没有个性化配置。开发环境、测试环境及生产环境无差异。

- 可替换。任意节点都能够被轻易地创建、销毁、替换。
- 自动化。所有的操作都通过工具自动化完成，无须人工干预。
- 可视化。当前环境要做到可控，就需要对当前的环境状况可视。
- 可追溯。所有的配置统一作为代码进行版本化管理，所有的操作都可以追溯。
- 快速。资源申请及释放要求秒级完成，以适应弹性伸缩和故障切换的要求。

目前比较常用的基础设施自动化工具包括 Ansible、Chef、SaltStack、Terraform 等，可以使用 DSL 定义、描述环境，更容易理解和维护。

容器将敏捷基础设施带到了另外一个高度，以下我们就通过容器建立敏捷基础设施。

### 3.3.1 容器 VS 虚拟机

容器和虚拟机到底有什么不同？这似乎是我被问到最多的一个问题。虽然二者功能类似，但是容器和虚拟机的差距非常大，它们的共同点只是都提供隔离环境。

容器和虚拟机主要的区别，如图 3-1 所示。虚拟机是在硬件的基础上进行虚拟化，隔离性更高，而容器是在操作系统上进行的虚拟化。严格意义上说，容器并不是虚拟化，因为所有容器都是共享内核的，也就是说，利用 Kernel 提供的隔离函数进行隔离。容器更像软件中的集装箱，能够把环境、配置、依赖、软件等封装起来。虚拟机多了一层 Guest OS，Guest OS 的变化不会影响底层的宿主操作系统，通过 Hypervisor 对底层基础设施进行抽象。下面从几个角度来分析一下容器和虚拟机。

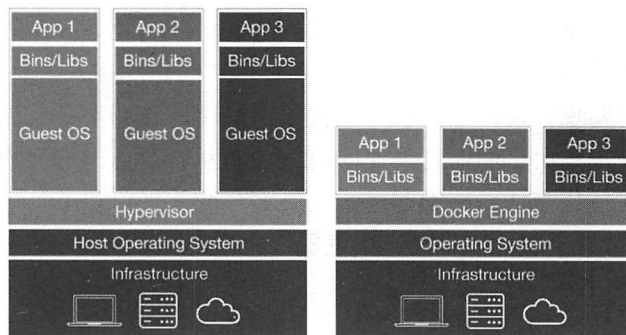


图 3-1 容器和虚拟机的主要区别<sup>①</sup>

- 资源利用率。从上面讲的容器和虚拟机的区别就能知道，容器更轻量级，占用的资源更少，比虚拟机的资源利用率高。当规模比较大的时候，这种节省还是很可观的。

<sup>①</sup> 图片来自 <https://blog.netapp.com/blogs/containers-vs-vms/>。



- 创建速度。Docker 的启动速度是秒级的，而虚拟机的启动速度一般是分钟级的，这是数量级上的差距。不要小看了这点差距，对于大规模的运维，每天都需要上线、回退、弹性伸缩很多次，这种大规模的资源调度，非常影响整体可用性。
- 性能。由于虚拟化需要运行完整的 Guest OS，不可避免地会出现性能损失，而容器相当于一个进程，性能相当于物理机。
- 隔离性。虚拟机隔离性更高，容器的隔离级别要低得多。

虚拟机相对物理机来说，通过统一资源调度平台（例如 OpenStack）提升了整体的资源利用率。例如，某电商平台的一台物理机运行着非常重要的业务，即使资源利用率很低，这台物理机也绝不允许部署其他业务，因为害怕受到影响。而虚拟化的隔离方式提升了整体的资源利用率，让资源可以流动起来。

容器的意义在于相对于物理机性能损失不大的情况下，为我们提供了标准化的运行环境，能够把复杂的配置封装到镜像中。特别是在微服务架构中，服务数量较多，上线频繁，为了保证高可用，需要轻量级的隔离机制，当流量变化时，需要快速伸缩。并且基于 Docker 的一整套生态非常健全，极大地提升了整体的效率。容器的高速度、敏捷性、可移植性使微服务架构更容易。

到目前为止，还有很多传统企业在面对容器的时候止步不前。从技术角度讲，小规模部署的时候，容器的隔离性差，安全性低，故障率会高一点，而大规模部署不依赖于单节点的可靠性，对这个问题并不敏感。并且，这也不仅仅是技术的原因。例如，某大型企业拥有自己的私有云，所有的物理机都实现了虚拟化，所有的业务应用都交给第三方开发，当然这个第三方是由很多企业组成的。在这种场景下，隔离性和安全性非常重要，所有的业务应用都要求不受其他应用影响，明确责任。当某个服务发生故障的时候，绝对不允许影响其他业务，所以在这里虚拟机隔离的不只是运行环境，还有“责任”。

看到这里，你可能仍然是一头雾水，这里面的精妙之处绝非几段文字可以描述清楚的，下面通过示例来讲解基于 Docker 的开发、测试、部署、运维流程。

### 3.3.2 安装 Docker

我们先了解一下 Docker，Docker 是一个开源的应用容器引擎，基于 Go 语言实现，并遵从 Apache 2.0 协议开源。到目前为止，Docker 已经成为容器的事实标准，应用十分广泛。Docker 可以让开发者打包应用及依赖包到一个轻量级、可移植的容器中，然后发布到任何安装了 Docker 的物理机或者虚拟机上，而不必担心是否安装了依赖项，不必考虑编译器或其他任何需要支持的基础设施。

如果要安装 Docker，可以到官网<sup>①</sup>下载安装包，因为我使用的是 MacBook 笔记本电脑，所以下载了 Mac 版本的 Docker 进行安装，也可以选择基于 Homebrew 进行安装，安装过程比较简单。安装成功后，在 MacBook 笔记本电脑右上方的工具栏中可以看到 Docker 的图标，如图 3-2 所示，点击对应项可以进行相应的设置。

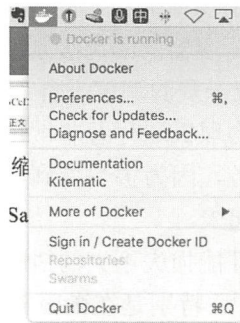


图 3-2 MacBook 下的 Docker 图标

也可以启动命令行，在命令行输入 `docker version` 命令进行验证，运行后会看到如图 3-3 所示的结果，证明安装成功，版本为 17.12.0-ce。

```
hl:opt root# docker version
Client:
 Version:      17.12.0-ce
 API version:  1.35
 Go version:   go1.9.2
 Git commit:   c97c6d6
 Built: Wed Dec 27 20:03:51 2017
 OS/Arch:     darwin/amd64

Server:
 Engine:
  Version:     17.12.0-ce
  API version: 1.35 (minimum version 1.12)
  Go version:  go1.9.2
  Git commit:  c97c6d6
  Built:       Wed Dec 27 20:12:29 2017
  OS/Arch:     linux/amd64
  Experimental: true
```

图 3-3 `docker version` 命令的执行结果

执行 `docker images` 命令可以查看本地镜像，如图 3-4 所示。

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
localhost:5000/docker-eureka	2.0	4bbf192c50ea	3 months ago	689MB
localhost:5000/docker-provider	2.0	8c0dc543e833	3 months ago	685MB
elasticsearch	latest	cb0029f9390c	4 months ago	581MB
openjdk	8-jre	e956268fd4ed	4 months ago	538MB

图 3-4 `docker images` 命令的执行结果

① 下载地址 <https://store.docker.com/editions/community/docker-ce-desktop-mac>。

### 3.3.3 部署私有 Docker Registry

在国内直接访问 Docker Hub 网速比较慢，拉取镜像的时间比较长，有两种方法可以解决这个问题，一是通过 VPN 加速，二是直接从国内的一些平台镜像仓库上拉取镜像。

在 macOS 系统下修改国内镜像仓库地址非常简单，只需要依次在任务栏点击 Docker for mac 应用图标→Perferences→Daemon→Registry mirrors，在列表中填写加速器地址即可。修改完成之后，按“Apply & Restart（应用配置并重启）”按钮，就可以使用新的镜像仓库地址了。

国内使用的比较多的镜像地址如下。

- 网易云镜像 <http://hub-mirror.c.163.com>。
- Daocloud 镜像 <https://hub.daocloud.io>。
- 阿里云镜像 <https://xxx.mirror.aliyuncs.com>，注意，阿里云镜像需要注册，xxx为专属域名。

除了慢的问题，还有代码安全性的问题，如果某些镜像不希望被外部访问，那就需要在本地建立镜像仓库，下面讲解如何在本地建立私有镜像仓库。

通过如下命令运行 Registry，如果本地没有 Registry 的镜像，则可以到上面我们配置的镜像市场中下载，如果不配置，则默认到 Docker Hub 中下载。

```
docker run -d -p 5000:5000 --name registry registry:2
```

然后，可以尝试通过本地镜像打标签，上传 Registry 进行验证。

```
docker tag ubuntu localhost:5000/Ubuntu
docker push localhost:5000/Ubuntu
```

运行 `docker images` 命令就可以看到刚刚上传到镜像仓库中的镜像了，如图 3-5 所示。

ubuntu	latest	0458a4468cbc	4 weeks ago	112MB
localhost:5000/ubuntu	latest	0458a4468cbc	4 weeks ago	112MB

图 3-5 docker images 命令的执行结果

到了这一步，后面所有的部署都可以基于镜像进行了，只要安装了 Docker，就可以很容易地部署服务。

### 3.3.4 基于 Spring Boot、Maven、Docker 构建微服务

3.3.3 节演示了如何从公共库中下载镜像并上传到本地私有库中，那么如何开发一个服务，并且将服务打包、制作镜像并上传到本地私有镜像库呢？下面演示一下如何通过 `docker-maven-plugin` 构建镜像。



根据前面的示例，假设已经通过 Eureka 建立了注册中心，分别开发了提供者和服务者，那么现在我们就改造一下提供者，把提供者打包为镜像。

先来改造一下项目的目录结构，如图 3-6 所示。这是我比较喜欢的一种目录结构，大家可以根据个人喜好建立。conf 目录下放置配置文件，不建议把配置文件放到 JAR 包里，因为放在 JAR 包里不方便查看、调试，src 放置源代码，assembly 用于放置打包需要的脚本及配置文件。

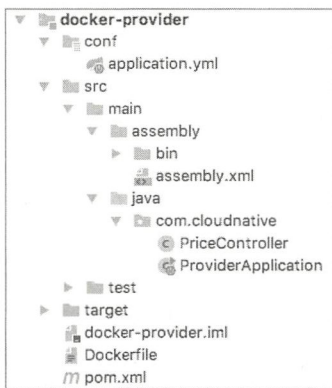


图 3-6 项目目录结构

通过 maven-assembly-plugin 插件进行打包，可以很容易地指定需要添加的文件和需要排除的文件，还可以设定 main 函数。以下代码为 maven-assembly-plugin 需要在 Maven 的配置文件 pom.xml 中进行的配置。

```
<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <configuration>
    <descriptor>src/main/assembly/assembly.xml</descriptor>
    <archive>
      <manifest>
        <mainClass>com.cloudnative.ProviderApplication</mainClass>
      </manifest>
    </archive>
  </configuration>
  <executions>
    <execution>
      <id>make-assembly</id>
      <phase>package</phase>
      <goals>
        <goal>single</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

```

    </execution>
  </executions>
</plugin>

```

配置文件 `pom.xml` 指定 `assembly.xml` 为描述文件，`assembly.xml` 描述了打包的格式及文件的移动位置。`maven-assembly-plugin` 支持的打包格式有 `zip`、`tar`、`tar.gz`（或者 `tgz`）、`tar.bz2`（或者 `tbz2`）、`jar`、`dir`、`war`，可以同时指定多种打包格式。`fileSets` 用来管理文件存放的位置，并且可以指定文件的权限，默认 `0644`。`dependencySets` 可以将 `scope` 为 `runtime` 的依赖包打包到 `lib` 目录下。

```

<assembly>
  <id>assembly</id>
  <formats>
    <format>dir</format>
  </formats>
  <includeBaseDirectory>true</includeBaseDirectory>
  <fileSets>
    <fileSet>
      <directory>src/main/assembly/bin</directory>
      <outputDirectory>bin</outputDirectory>
      <fileMode>0755</fileMode>
    </fileSet>
    <fileSet>
      <directory>conf</directory>
      <outputDirectory>conf</outputDirectory>
      <fileMode>0644</fileMode>
    </fileSet>
  </fileSets>
  <dependencySets>
    <dependencySet>
      <outputDirectory>lib</outputDirectory>
    </dependencySet>
  </dependencySets>
</assembly>

```

打包后的项目目录结构，如图 3-7 所示。`bin` 为 `shell` 脚本，`conf` 为配置文件，`lib` 是项目的 `jar` 和依赖的 `jar` 文件。

`docker-maven-plugin` 有两种配置方式，一种不需要写 `dockerfile`，直接在 `pom` 中完成配置，另外一种需要写 `dockerfile`。我认为通过 `dockerfile` 的方式更灵活，本节以写 `dockerfile` 的方式进行配置。



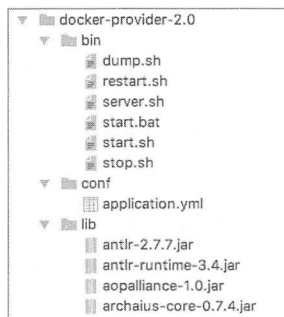


图 3-7 项目目录结构

在 pom.xml 中配置 docker-maven-plugin 构建信息。我们使用 Spotify 提供的 plugin，最新版本为 1.0.0，相应说明可以看注释。

```
<plugin>
  <groupId>com.spotify</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <version>1.0.0</version>
  <executions>
    <execution>
      <id>build-image</id>
      <phase>package</phase>
      <goals>
        <goal>build</goal>
      </goals>
      <configuration>
        <dockerDirectory>${project.basedir}</dockerDirectory>
        <!-- 镜像名称 -->

<imageName>${docker.registry}/${project.artifactId}:${project.version}</imageName>
      </configuration>
    </execution>
  </executions>
  <configuration>
    <!-- 私有仓库配置，需要在 Maven 的 settings.xml 文件配合 serverId 对应 -->
    <serverId>wj</serverId>
    <registryUrl>localhost:5000/v2</registryUrl>
    <!-- 执行 mvn install 阶段上传，否则只有 deploy 阶段上传 -->
    <pushImage>true</pushImage>
    <forceTags>true</forceTags>
    <imageName>
      ${docker.registry}/${project.artifactId}:${project.version}
```



```

    </imageName>
    <imageTags>
      <!--tag 为项目版本号-->
      <imageTag>2.0</imageTag>
    </imageTags>
  </configuration>
</plugin>

```

在项目根目录下建立 `dockerfile` 文件，下面只是简单声明了基础镜像，将 `target/docker-provider-2.0-assembly/docker-provider-2.0` 目录添加到基础镜像中，一起打包，最后通过 `start.sh` 启动项目。

```

FROM java:8
ADD target/docker-provider-2.0-assembly/docker-provider-2.0 /docker-provider-2.0
ENTRYPOINT ["sh", "/docker-provider-2.0/bin/start.sh"]

```

由于项目要在容器中启动，因此需要修改一下 `aplication.yml` 的注册中心地址，最简单的方式是直接写真实的 IP 地址。

```
defaultZone: http://172.20.101.32:10001/eureka
```

执行 `mvn clean install` 命令，可以看到如下日志信息。

```

[INFO] Building image localhost:5000/docker-provider:2.0
Step 1/3 : FROM java:8
--> d23bdf5b1b1b
Step 2/3 : ADD target/docker-provider-2.0-assembly/docker-provider-2.0
/docker-provider-2.0
--> 7bclc0dd1b0c
Step 3/3 : ENTRYPOINT ["sh", "/docker-provider-2.0/bin/start.sh"]
Successfully built 3218c2b9a5e6
Successfully tagged localhost:5000/docker-provider:2.0
[INFO] Built localhost:5000/docker-provider:2.0
[INFO] Tagging localhost:5000/docker-provider:2.0 with 2.0
[INFO] Pushing localhost:5000/docker-provider:2.0
The push refers to repository [localhost:5000/docker-provider]
13282abe3b97: Pushed

```

执行 `docker images` 命令，可以看到镜像已经推到仓库。

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
localhost:5000/docker-provider	2.0	ff98b187d3ed	22 seconds ago	685MB

执行容器启动命令。

```
docker run -i -t -p 10004:10004 localhost:5000/docker-provider:2.0
```

看到如下日志，表示已经启动成功。

```
Updating port to 10004
Started ProviderApplication in 4.888 seconds (JVM running for 5.41)
```

执行 `docker ps` 命令查看运行情况，正常启动。

```
4e4alfaba74b          localhost:5000/docker-provider:2.0    "sh /docker-provider..."
About a minute ago    Up About a minute    0.0.0.0:10004->10004/tcp    reverent_payne
```

通过 URL “`http://localhost:10004/price/20`” 得到结果 `{id:20}`。

### 3.3.5 基于 docker-compose 管理容器

批量部署容器如果用上面的方法就比较麻烦了，`docker-compose` 正是用来部署自动化的工具，相当于通过脚本把一些零碎的工作串联起来了。例如要启动容器、设置磁盘映射、设置 NAT 网络或者桥接网络等需要大量的调试，这显然是非常低效的，也不符合敏捷基础设施的理念。我们只要定义一个 `docker-compose.yml` 文件，运行命令 `docker-compose up -d` 就可以了。

`docker-compose` 安装非常简单，在 macOS 系统下直接用 `pip` 命令或者 `brew` 命令就可以安装。

例如，运行上面制作的 `docker-provider` 镜像，要先编写一个 `docker-compose.yml` 文件即可，详细信息如下。

```
eureka-server:
  image: localhost:5000/docker-eureka:2.0
  hostname: eureka-server
  ports:
    - "10001:10001"

service-provider:
  image: localhost:5000/docker-provider:2.0
  depends_on:
    - eureka-server
  ports:
    - "10044:10004"
  links:
    - "eureka-server"
```

以上分别定义了两个容器，一个是注册中心，一个是提供者，注册中心端口为 10001，与宿主机 10001 端口映射，也就是说访问宿主机的 10001 端口，会转发到容器中的 10001

端口。当然，可以映射多个端口。另外，Links 参数可以确保两个容器之间的通信，例如 provider 链接另一容器服务 Eureka，可以给出服务名和别名，也可以仅给出服务名，这样别名将和服务名相同，效果同 docker run -link.depends\_on 用于指定服务依赖，例如 MySQL、Redis 等。如果指定了依赖，则会优先于服务创建并启动依赖。

运行 docker-compose up -d 命令就可以分别启动两个容器，如图 3-8 所示。

```
hl:opt root# docker-compose up -d
Starting opt_eureka-server_1 ... done
Starting opt_service-provider_1 ... done
```

图 3-8 docker-compose up -d 命令的执行结果

运行 docker-compose logs 命令可以查看日志。

使用命令 docker-compose ps 查看运行状况，如图 3-9 所示。

```
hl:opt root# docker-compose ps
```

Name	Command	State	Ports
opt_eureka-server_1	sh /docker-eureka-2.0/bin/ ...	Up	0.0.0.0:10001->10001/tcp
opt_service-provider_1	sh /docker-provider-2.0/bi ...	Up	0.0.0.0:10044->10004/tcp

图 3-9 docker-compose ps 执行结果

此时，在宿主机通过浏览器访问 <http://localhost:10044/price/20> 可以看到结果返回 {id:20}。

如果要终止运行，可以直接运行 docker-compose stop 命令，如图 3-10 所示。

```
hl:opt root# docker-compose stop
Stopping opt_service-provider_1 ... done
Stopping opt_eureka-server_1 ... done
```

图 3-10 docker-compose stop 命令的执行结果

### 3.4 基于公共基础服务的平台化

平台化是指利用公共基础服务提升整体架构能力的模式。公共基础服务是指与业务无关的、通用的服务，包括监控服务、缓存服务、消息服务、数据库服务、负载均衡、分布式协调、分布式任务调度等。

一个比较典型的重视平台化的公司是 Google。Google 相当重视平台化战略，在很多年以前就开始重点建设平台，Google 招的工程师并不比其他公司强多少，但是 Google 强大的平台让他们在业务开发的时候得心应手。国内的很多互联网公司也是一样，如阿里巴巴、腾讯等，很早就开始重视平台化。

平台可以提升团队的下限，让平凡的业务开发人员做出不平凡的系统。平台化的核心思想也就是把复杂的、通用的逻辑部分统一抽象到一个地方，让最核心的工程师去研发，



以保证高效、稳定。例如数据迁移服务，可能很多系统重构时都会用到，那么让一个团队专门做这件事情，好于让每个重构的团队都经历一遍。因为他们重复做这件事，不断积累经验，然后不断优化迁移服务，可以让整个事情变得更简单、快速、可靠。

通常很多大型互联网公司的平台部门是由骨干研发人员组成的，是整个公司技术的核心，架构师将自己的架构思想及理念灌注到平台当中，使业务系统能够在正确的轨道上运行。

由于篇幅所限，无法详细描述所有公共基础服务，下节介绍几个常用的公共基础服务。

### 3.5 监控告警服务

无论什么架构监控，告警服务都是不可或缺的，否则以下场景将让你头疼不已。

- 每次上线都提心吊胆，总是害怕出现问题。
- 用户比我们更快发现故障，并通过客服告诉我们。
- 一旦出现故障，花费大量时间、人力定位问题。
- 总是怀疑网络是不是有问题，磁盘是不是有问题。

监控告警永远是系统运维最重要的话题。要实现快速交付，就必须在自动化测试、灰度发布、监控告警、故障定位及修复上面下功夫。其中监控能让系统处于可控的情景中，故障出现时，快速发现并报警，可以有效提升可用性。

不只是发生故障的告警，当系统达到某个阈值，可能要发生故障的时候，我们也希望能够快速收到通知。海恩法则<sup>①</sup>指出：每一起严重事故的背后必然有 29 次轻微事故和 300 起未遂先兆，以及 1000 起事故隐患。法则强调两点：一是事故的发生是量的积累的结果；二是再好的技术，再完美的规章，在实际操作层面，也无法取代人自身的素质和责任心。我们希望实时看到关于生产环境甚至包括预发布环境的系统状况，希望实时看到各类硬件指标，如 CPU 占用率、内存、磁盘、带宽等，当然，也包括服务的重要指标，如错误率、响应时间、吞吐量、消息堆积数等，以此来了解目前系统的健康状态，到安全水位线的距离，这是系统进行伸缩、限流的重要依据。

传统的监控方式更强调以资源为中心，如关注 CPU、内存、带宽等，这种监控方式显然不够精确，容易导致误报。目前监控的趋势逐步向应用一侧倾斜，如订单量突然下降一半是否存在问题；吞吐量到达阈值后先关注依赖的其他系统是否有问题，再弹性伸缩，我们称之为“以应用为中心的监控”。

---

<sup>①</sup> 海恩法则（Heinrich's Law），是飞机涡轮机的发明者德国人帕布斯·海恩提出的一个在航空界关于安全飞行的法则。

### 3.5.1 监控数据采集

作为监控系统，一切都需要基于数据，然后才能进行各种分析、处理、报警，所以采集数据是监控系统非常重要的一步。常用的监控数据采集方式有以下三种。

(1) 直接上报，如图 3-11 所示。在物理机、虚拟机或容器里，应用通过 SDK 直连监控服务，直接发送或通过监控拉取相应的监控数据。

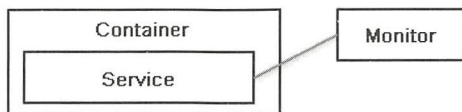


图 3-11 直接上报

(2) 通过日志上报，如图 3-12 所示。在物理机、虚拟机或容器里，应用通过 SDK 打印规定格式的日志，打印到指定目录，监控系统通过日志收集组件收集相应的信息，并且做大数据分析，发送给监控系统。

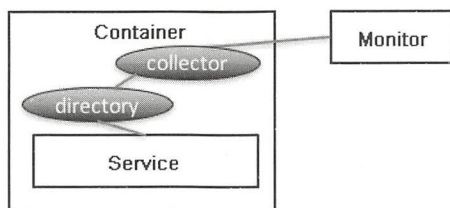


图 3-12 通过日志上报

(3) 通过 agent 上报，如图 3-13 所示。在物理机、虚拟机或容器里，监控系统会在镜像中默认安装一个 agent，用来获取各种系统的监控指标。

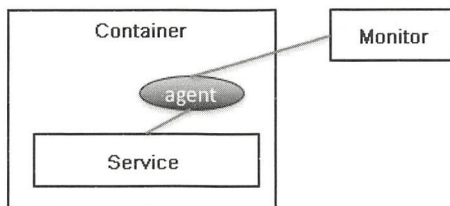


图 3-13 通过 agent 上报

### 3.5.2 监控数据接收模式

监控数据的接收模式主要分为两种，分别是推（push）模式和拉（pull）模式，如



图 3-14 所示。这两种模式的主要区别是 API 放在哪一方，被调用方需要提供 API 接口。

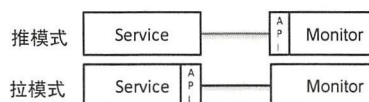


图 3-14 推模式和拉模式

### (1) 推模式。

业务服务主动通过 Monitor 的 API 与其建立连接，当产生数据时，通过连接直接发送数据。

### (2) 拉模式。

业务服务到 Monitor 上注册标准的数据收集 API 接口和相应配置（包括调用时间、频率等），Monitor 根据配置调用业务服务的 API 获取数据。

## 3.5.3 通过时间序列数据库存储监控数据

一方面，由于监控系统的应用很广泛，导致了监控系统要采集的数据种类很多，例如 CPU、内存、磁盘、数据库、缓存等，这些数据各有各的特点，很难抽象到一种数据结构，传统的监控方式将意味着巨大的工作量。另一方面，微服务架构的大规模应用，快速且频繁的变更，对可用性的要求提高需要我们快速定位故障并解决，传统的监控无法存储这么大的数据。

新一代的监控系统基于时间进行了抽象，我们发现，所有监控数据图表的背后都包含了时间，一般横轴是时间，纵轴是数值。基于时间序列存储监控数据是目前大型互联网公司的一个普遍做法，例如 Google、百度等。

图 3-15 是著名数据库 DB-Engines 中的统计数据，可以看到，目前比较流行的时序数据库包括 InfluxDB、OpenTSDB、Druid、Graphite 等。

时序数据库的特点如下。

- 写多读少，大部分时间是写入，写入是顺序的，但是读的时候并发量也有可能很高。
- 数据量较大，内存一般放不下，属于 IO 密集型。
- 读操作的时候需要按照时间排序，升序或降序。

## 3.5.4 开源监控系统实现 Prometheus

Prometheus 是一套开源的监控、报警、时间序列数据库的组合，最初由 SoundCloud 公司开发。Prometheus 在 2016 加入 CNCF (Cloud Native Computing Foundation)，成为继





Kubernetes 之后的第二个由基金会主持的项目。目前社区非常活跃，有高达 1.4 万个 Star。Prometheus 非常适合监控 Kubernetes。

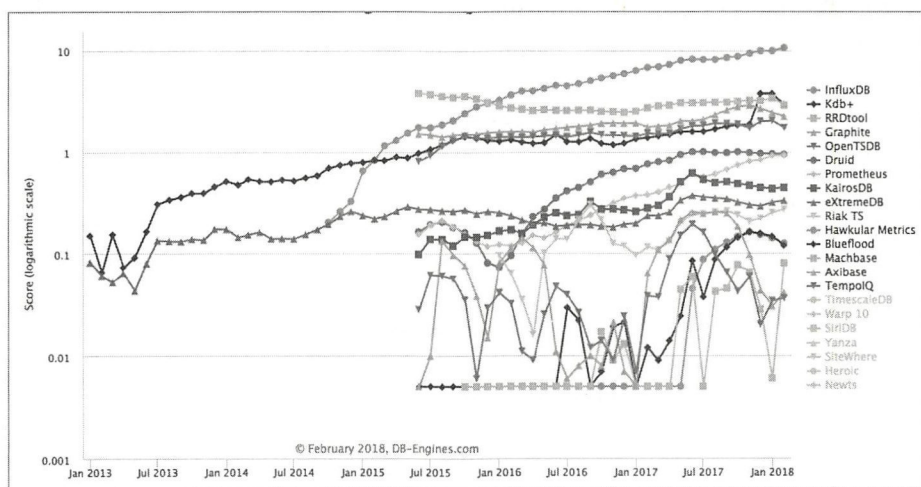


图 3-15 DB-Engines 中的统计数据<sup>①</sup>

与一般的监控系统相比，Prometheus 具有如下特点。

- 时序数据库存储监控数据能够存储更大量的数据，不依赖于其他存储系统，安装非常简单。
- 灵活的查询语言（PromQL）。
- 通过基于 HTTP 的 pull 方式采集时序数据，可以通过 Pushgateway 进行时序数据推送。
- 多种可视化和仪表盘支持。

Prometheus 的架构，如图 3-16 所示。

Prometheus Server 主要由三部分组成。

- Retrieval: 负责定时去指定的 API 上抓取采样指标数据。
- Storage: 负责将采样数据高效安全地持久化存储到磁盘中。
- PromQL: 负责提供查询。

Prometheus Server 端会存储所有的数据，并对这些数据进行分析，基于规则进行报警。Prometheus 利用自身的时序数据库，有着非常高效的存储。据官方介绍，每个采样数据只占 3.5byte 左右，上百万条时间序列，30s 间隔，保留 60 天，大概只占用 200GB。Prometheus

<sup>①</sup> 图片来自 [https://db-engines.com/en/ranking\\_categories](https://db-engines.com/en/ranking_categories)。



支持通过配置文件、文本文件、ZooKeeper、Consul、DNS SRV lookup 等方式指定抓取目标。Prometheus 是通过主动获取的方式采集数据，但是客户端有两种方式输送数据，一种方式是业务服务提供 HTTP 接口，Prometheus 定时从业务服务中获取数据；另外一种方式是业务服务推送数据到 Pushgateway，Prometheus 定时从 Pushgateway 中获取数据。

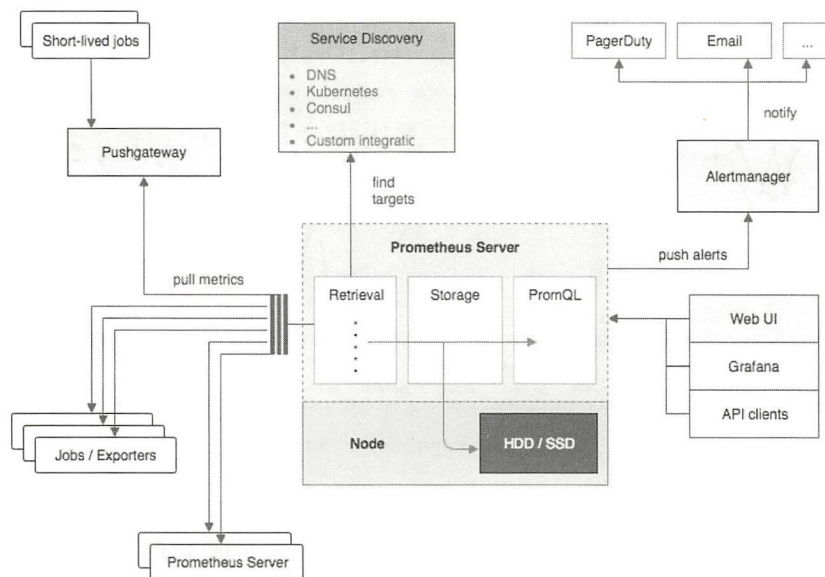


图 3-16 Prometheus 架构<sup>①</sup>

Alertmanager 是独立于 Prometheus 之外的一个组件，提供十分灵活的报警方式。另外，可以通过 Web UI、Grafana、API clients 等可视化界面查询数据。

Prometheus 支持如下四种数据类型。

- Counter: 单调递增数据，只能增加，不能减少。例如记录请求次数、错误数。
- Gauge: 当前的状态，常规值，可增可减。例如内存、CPU 的使用情况。
- Histogram: 主要用于在指定分布范围内（Buckets）记录大小或者事件发生的次数。
- Summary: 客户端定义的数据分布统计图。

### 3.5.5 通过 Prometheus 和 Grafana 监控服务

实践证明，Prometheus 和 Grafana 是一对完美的组合，把它们集成起来非常简单。在 macOS 系统上直接输入 `brew install` 就可以安装成功。通过 `./prometheus--config.file=`

<sup>①</sup> 图片来自 <https://prometheus.io>。



prometheus.yml 命令启动 Prometheus，访问 <http://localhost:9090> 也可以直接看到 Prometheus 的界面，如图 3-17 所示。



图 3-17 Prometheus 的管理界面

在 Prometheus 的配置文件 prometheus.yml 中添加监控任务，假设我们要监控的服务的目标地址为 localhost:10002，监控任务的名字为 app，metrics 的路径为/prometheus，此配置要和服务中的配置相对应，配置文件如下所示。

```
- job_name: 'app'
  metrics_path: '/prometheus'
  static_configs:
    - targets: ['localhost:10002']
      labels:
        instance: app
```

在原有的 eureka-provider 中配置 pom.xml 文件，增加 Prometheus 收集日志所需的 JAR 包。

```
<dependency>
  <groupId>io.prometheus</groupId>
  <artifactId>simpleclient_spring_boot</artifactId>
  <version>0.1.0</version>
</dependency>
<dependency>
  <groupId>io.prometheus</groupId>
  <artifactId>simpleclient_hotspot</artifactId>
  <version>0.1.0</version>
</dependency>
<dependency>
  <groupId>io.prometheus</groupId>
  <artifactId>simpleclient_servlet</artifactId>
```





```
<version>0.1.0</version>
</dependency>
```

利用 Spring MVC 定义拦截器。

```
public class RequestCounterInterceptor extends HandlerInterceptorAdapter {

    private static final Counter requestTotal = Counter.build()
        .name("http_requests_total_spring")
        .labelNames("method", "handler", "status")
        .help("Http Request Total").register();

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception e) {
        String handlerLabel = handler.toString();
        if (handler instanceof HandlerMethod) {
            Method method = ((HandlerMethod) handler).getMethod();
            handlerLabel = method.getDeclaringClass().getSimpleName() + "." +
method.getName();
        }
        requestTotal.labels(request.getMethod(), handlerLabel, Integer.toString
(response.getStatus())).inc();
        System.out.println ("request = [" + request + "], response = [" + response +
"], handler = [" + handler + "], e = [" + e + "]);
    }
}
```

编写配置类，通过@Configuration 表明身份。添加拦截器，并且拦截所有请求。

```
/**
 * 配置类
 */
@Configuration
public class MonitorConfig extends WebMvcConfigurerAdapter {

    @Bean
    public SpringBootMetricsCollector springBootMetricsCollector(Collection<
PublicMetrics> publicMetrics) {
        SpringBootMetricsCollector springBootMetricsCollector = new
SpringBootMetricsCollector(publicMetrics);
        springBootMetricsCollector.register();
        return springBootMetricsCollector;
    }

    @Bean
    public ServletRegistrationBean servletRegistrationBean() {
```



```
DefaultExports.initialize();
//在 Prometheus 中设置路径，在 yml 中需要对应
return new ServletRegistrationBean(new MetricsServlet(), "/prometheus");
}
@Override
public void addInterceptors(InterceptorRegistry registry) {
    //addPathPatterns 添加连接器并拦截所有请求
    registry.addInterceptor(new
RequestCounterInterceptor()).addPathPatterns("/**");
    super.addInterceptors(registry);
}
}
```

启动类需要通过@EnablePrometheusEndpoint 启用 Prometheus endpoint。

```
@EnableDiscoveryClient
@SpringBootApplication
@EnablePrometheusEndpoint
public class ProviderApplication implements CommandLineRunner {

    @Autowired
    private DiscoveryClient discoveryClient;

    public String serviceUrl() {
        List<ServiceInstance> list = discoveryClient.getInstances("PROVIDER");
        if (list != null && list.size() > 0 ) {
            return String.valueOf(list.get(0).getUri());
        }
        return null;
    }

    public static void main(String[] args) {
        SpringApplication.run(ProviderApplication.class,args);
    }

    @Override
    public void run(String... strings) throws Exception {
        DefaultExports.initialize();
    }
}
```

启动服务，访问 <http://localhost:10002/prometheus> 目录会发现已经有了监控指标，如图 3-18 所示。



```
localhost:10002/prometheus

Prometheus Time Series... localhost:10002/metrics Grafana - Prometheus 2.0... localhost:10002/price/212 Eureka

# HELP process_cpu_seconds_total Total user and system CPU time spent in seconds.
# TYPE process_cpu_seconds_total counter
process_cpu_seconds_total 6.22416
# HELP process_start_time_seconds Start time of the process since unix epoch in seconds.
# TYPE process_start_time_seconds gauge
process_start_time_seconds 1.519974964222E9
# HELP process_open_fds Number of open file descriptors.
# TYPE process_open_fds gauge
process_open_fds 174.0
# HELP process_max_fds Maximum number of open file descriptors.
# TYPE process_max_fds gauge
process_max_fds 10240.0
# HELP counter_servo_eurekaclient_transport_request_counter_servo_eurekaclient_transport_request
# TYPE counter_servo_eurekaclient_transport_request gauge
counter_servo_eurekaclient_transport_request 0.0
# HELP counter_servo_eurekaclient_transport_request_counter_servo_eurekaclient_transport_request
# TYPE counter_servo_eurekaclient_transport_request gauge
counter_servo_eurekaclient_transport_request 0.0
# HELP normalized_servo_eurekaclient_transport_latency_totaltime_normalized_servo_eurekaclient_transport_latency_totaltime
# TYPE normalized_servo_eurekaclient_transport_latency_totaltime gauge
normalized_servo_eurekaclient_transport_latency_totaltime 0.0
# HELP normalized_servo_eurekaclient_transport_latency_count_normalized_servo_eurekaclient_transport_latency_count
# TYPE normalized_servo_eurekaclient_transport_latency_count gauge
```

图 3-18 监控指标

访问 <http://localhost:10002/price/1> 做测试，到 <http://localhost:9090/graph> 路径下添加 `http_requests_total_spring` 并执行可以看到统计结果，如图 3-19 所示。



图 3-19 统计结果





安装 Grafana 后添加 Prometheus 作为数据源，按“Add Query”按钮添加要监控的指标，如图 3-20 所示。

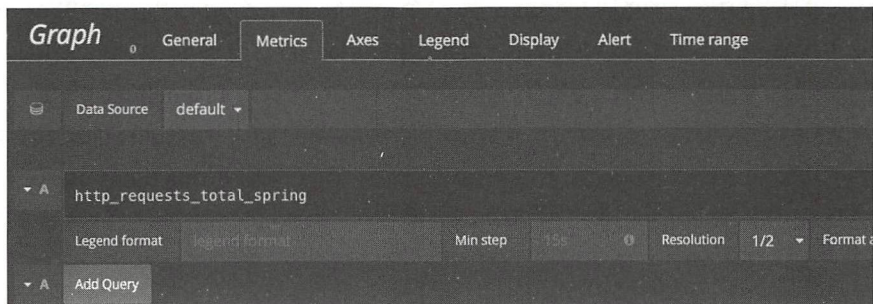


图 3-20 为 Grafana 添加 Prometheus

经过一段时间，Grafana 就可以通过图表展示监控指标了，如图 3-21 所示。



图 3-21 Grafana 统计结果展示

注意，Prometheus 的启动配置存在优化空间，对性能影响较大。最好将 Prometheus 安装到一个独立的容器内，否则，需要限制 CPU 和内存的占用。另外需要注意启动参数，`-storage.local.target-heap-size 2147483648` 表示 Prometheus 独占的内存空间，默认是 2GB 的内存空间，这个参数非常重要，官方文档中建议配置为可用内存的三分之二，Prometheus 可以直接将数据加载到内存中，让计算查询变得更快。`-storage.local.retention 8760h0m0s` 表示保存 1 年的数据，可以根据场景灵活配置数据保存的时间，监控数据是非常占用存储空间



间的，可以结合这个参数控制磁盘占用。

虽然单机版 Prometheus 已经足够强大，但是在大规模场景中仍然需要扩展。Prometheus 有两个比较好的扩展版，一个是 Cortex，另一个是 Thanos。Cortex 是一个 Weaveworks 项目，它构成了 Weave Cloud 的监控后端。要使用 Cortex 需要先注册 Weave Cloud。Thanos 被定义为一组通过跨集群联合、跨集群无限存储和全局查询为 Prometheus 增加高可用性的组件，利用 V3 引擎的特性，支持将历史数据存储到对象存储中。Thanos 的架构，如图 3-22 所示。例如 AWS 的 S3，Thanos 引入了一个 Sidecar 节点，和 Prometheus Server 部署到一起，将本地数据传输到远程的对象存储中。为了支持查询的时候能够查到远程的数据和本地的数据，又引入了 Store 层来屏蔽底层的对象存储细节。

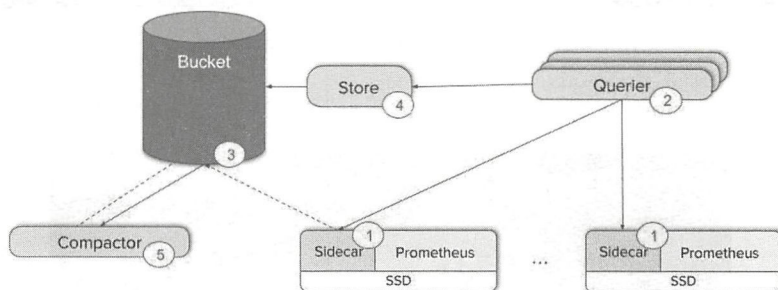


图 3-22 Thanos 的架构<sup>①</sup>

## 3.6 分布式消息中间件服务

分布式消息中间件是公共基础服务中非常重要的一个服务，在大规模、高并发架构中扮演着非常重要的角色。

两个服务之间的通信方式有两种，一种是同步调用，另外一种异步调用，虽然异步的方式可以极大提升吞吐量，但是异步调用是有成本的，开发、调试起来相对复杂。

如果我们在两个服务之间放一个 Broker，Producer 把消息投递到 Broker 之后就返回，由 Broker 确保消息被 Consumer 消费成功，如图 3-23 所示。



图 3-23 生产者-消费者模式

<sup>①</sup> 图片来自 <https://improbable.io/games/blog/thanos-prometheus-at-scale>。



### 3.6.1 分布式消息中间件的作用

消息中间件在分布式系统中到底起到什么样的作用呢？

通过分布式消息中间件解耦。同步调用是一种强依赖，而异步调用是一种弱依赖，很多场景下强依赖会导致彼此互相影响，可用性下降。举个例子，业务需要在登录的过程中给用户加积分，如果采用同步调用，验证用户登录已经成功，但是由于加积分服务出现故障，会导致整个登录流程失败，这是一个非常典型的问题，如图 3-24 所示。

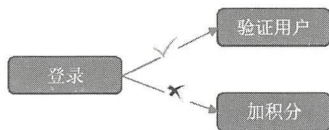


图 3-24 登录加积分过程

如果我们可以把加积分的动作通过分布式消息中间件传递给加积分服务，即使加积分服务挂掉，也不会影响登录动作。通过消息中间件，系统之间可以进行可靠的异步通信，从而降低系统之间的耦合度，系统能够得到更好的扩展性和可用性。

分布式消息中间件有削峰填谷的作用。在发微博服务突然出现大量请求的情况下（例如节假日祝福），如果采用同步调用，则信息流服务可能无法承受；如果没有限流，则会导致信息流服务大量超时，甚至宕机，而限流是有损的，会有一部分超出信息流服务能力的请求失败，如图 3-25 所示。在这种场景下使用分布式消息中间件可以起到削峰填谷的作用。



图 3-25 限流与消息中间件

通过分布式消息中间件降低响应时间。利用可靠事件模式，一旦发微博数据到达消息中间件，就认为发微博成功了，显然比直接持久化到数据库的响应时间要少得多。在这种场景下，至于关注者是否马上就可以收到这条消息并不是特别重要。

通过分布式消息中间件提升吞吐量。如果采用关系型数据库，则写的吞吐量比较低，这是底层的数据结构决定的，例如 MySQL。为了提升读性能，需要建立索引，而索引的数





据结构是 BTree，它在磁盘上随机写，性能比较低。即使不批量发送，在 ACK 设置为同步到所有 Follower 后返回三节点情况下，Kafka 的吞吐量是 MySQL 吞吐量的三四十倍，并且 Kafka 的扩展性远高于 MySQL。另外，在微服务架构下，往往一次请求不只是写一个数据库，在微博中，除了同步信息流，可能还要进行更新缓存、建库外索引、敏感词过滤等相关操作。

一般来说，数据库是整个系统中最难扩展的地方，因为它涉及数据迁移，常常成为系统扩展的瓶颈，消息中间件能够缓解数据库的瞬时压力。

### 3.6.2 业界常用的分布式消息中间件

目前业界应用比较多的分布式消息中间件主要包括：ActiveMQ、RabbitMQ、Kafka、RocketMQ，它们在各自的应用领域都有广大的用户群体，虽然都是分布式消息中间件，但是每种分布式消息中间件应用的方式还是有很大区别的。

#### ActiveMQ

优点：Apache 开源，历史悠久，功能丰富，能够适配各种协议，文档多，有鉴权机制，多语言客户端。

缺点：性能低，社区越来越不活跃，只支持主从，扩展性差。

#### RabbitMQ

优点：它可以看成是 ActiveMQ 的改进版，用 Erlang 语言实现，性能比 ActiveMQ 的高，功能丰富，支持的协议多（AMQP、XMPP、SMTP、STOMP）。

缺点：虽然性能比 ActiveMQ 的高，但是相比 Kafka、RocketMQ 还有差距。它只支持主从，扩展性差。

#### Kafka

优点：Apache 开源，性能非常高，0.8 版本以后消息可靠性得到了保障，应用范围大幅扩大，分布式能力强大，支持多语言客户端，文档丰富。

缺点：支持的协议少，管理工具少。

#### RocketMQ

优点：Apache 开源，Java 语言开发，模仿了 Kafka 的设计理念，继承了 Kafka 高性能、



分布式能力强的优点。同时，一些对企业应用比较好的功能非常有价值，例如，消息服务端过滤、定时消息等。

缺点：文档相对少，成功案例相对少。

本书将重点介绍 Kafka，Kafka 独特的设计理念使它成为分布式消息中间件中的佼佼者。

### 3.6.3 Kafka 的设计原理

Kafka 是一个分布式消息中间件，但是它并不符合 JMS 规范，即使消息已经被消费，也不会被马上删除，当消息保留一定时间后，会被批量删除，无论消息是否被消费过，用这种方式可以极大地降低磁盘的 IO。在 Kafka 中，消息被持久化到磁盘，因此，Kafka 堆积消息的能力非常强大。另外，Kafka 的扩展性极好，可以轻易地扩展出多个节点。但是到目前为止，Kafka 还需要依赖于 ZooKeeper 管理元数据。

如果要了解 Kafka，就得先看看 Kafka 中有哪些角色，Kafka 架构图展示出了各个角色之间的关系，如图 3-26 所示。

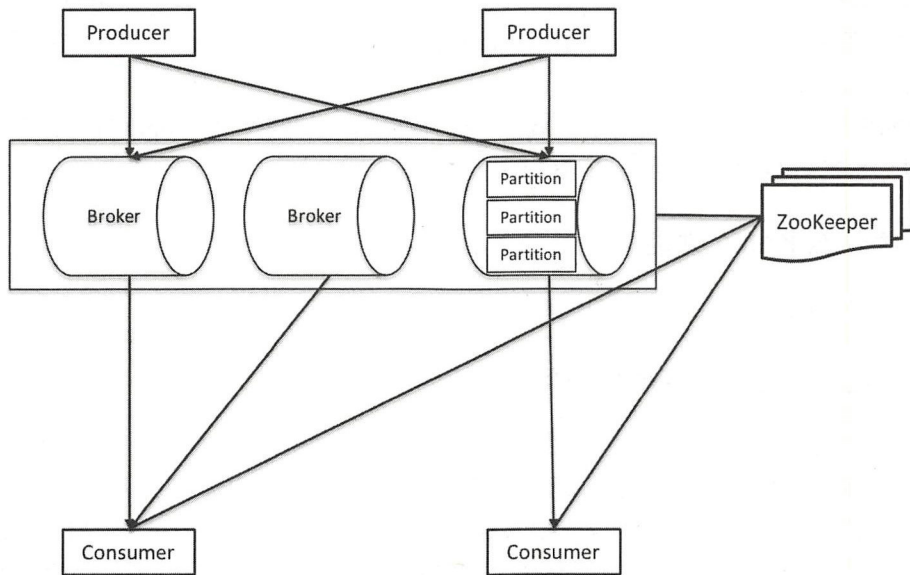


图 3-26 Kafka 架构图

- **Broker:** Kafka 的服务端，负责接收数据，并持久化数据，Broker 可以有多个，每个 Broker 可以包含多个 Topic，Broker 并不保存 Offset（消费者消费的位置）数据，由 Consumer 自己负责保存，默认保存在 ZooKeeper 中。



- **Producer:** 生产者。生产数据发送到 Broker 存储数据，需要注意的是，Producer 直连 Broker，不经过任何代理，Producer 将会和 Topic 下所有 Partition Leader 保持 socket 连接。Producer 还有异步发送的能力，也就是说，将多条消息缓冲到客户端，到达一定数量或时间后，批量发送给 Broker。通常 Producer 是一个包含 Kafka 客户端的业务服务。
- **Consumer:** 消费者。业务服务从 Broker 订阅 Topic，并从订阅的 Topic 中接收数据。每个消费者都属于某个消费者组，一个组里的消费者订阅的是同一个 Topic，同一个组的消费者分别订阅同一个 Topic 下的不同 Partition 的数据。需要注意的是，每个 Partition 只能被一个消费者订阅，一个消费者可以订阅多个 Partition，用这种方式避免一定的重复消费。如果认为消费者的能力受到了限制，则可以通过增加 Partition 的方式实现扩展，当一个消费者挂掉之后，会重新进行负载均衡。如果网络不稳定，则会导致频繁的重新负载均衡。通常 Consumer 是一个包含 Kafka 客户端的业务服务。
- **Topic:** 主题。相当于数据库中的表名，生产者和消费者之间通过 Topic 建立对应关系。Topic 更像一个逻辑概念，每个 Topic 下包含多个 Partition，所有的元数据都存储在 ZooKeeper 中。
- **Partition:** 分区。Kafka 为了扩展性，提升性能，可以将一个 Topic 拆分为多个分区，每个分区可以独立放到一个 Broker 上。

### 3.6.4 为什么 Kafka 性能高

为什么 Kafka 能获得如此高的性能呢？

#### 顺序写磁盘——媲美内存随机访问

操作系统并不提供物理内存直接给应用程序访问，而是使用虚拟内存访问机制。当发生大量内存随机访问时，磁盘随机访问速度可能不如直接顺序访问磁盘的速度快。磁盘随机访问速度之所以慢，是因为受限于磁头调度，一次磁头调度大概需要几毫秒到几十毫秒不等。使用 6 个 7200rpm SATA RAID-5 阵列的 JBOD 配置上的线性写入性能约为 600MB/s，但是随机写入的性能大约只有 100KB/s，两者之间有超过 6 千倍的差距。如果采用顺序写入，使用千兆网卡，那么写磁盘的速度会超过网卡的带宽，也就是说，网络将成为瓶颈点，磁盘不再是瓶颈点了。





## 零拷贝——减少上下文切换及拷贝次数

要弄明白零拷贝，就要先了解 Linux 操作系统的架构。从宏观上来看，Linux 操作系统的体系架构分为用户态和内核态（或者用户空间和内核空间）。

- 用户态：只能受限地访问内存，不允许访问外围设备。占用 CPU 的能力被剥夺，CPU 资源可以被其他程序获取。
- 内核态：CPU 可以访问内存中所有的数据，包括外围设备，例如硬盘、网卡，CPU 也可以将自己从一个程序切换到另一个程序。

这样做的主要原因是限制不同的程序之间的访问能力，防止它们获取别的程序的内存数据。

业务应用可以非常方便地利用用户态 buffer，也可以利用内核态 buffer。内核态 buffer 可以看作是磁盘或者网络的缓存，如图 3-27 所示。实际上利用业务应用管理内存是非常复杂的，如果使用 JVM，则采用堆内缓存将受到 GC 的惩罚，我们都知道发生 FullGC 的时候，程序对外停止响应。另外，如果数据量比较大，重建缓存也是非常耗时的（10GB 缓存可能需要 10 分钟）。

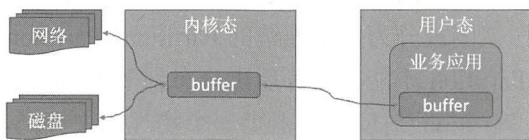


图 3-27 内核态缓存与用户态缓存

操作系统将数据从磁盘读取到内核空间中的 Page Cache 过程大致如下。

- (1) 业务应用从内核空间读取数据到用户空间缓冲区。
- (2) 业务应用将数据写入内核空间到 socket 缓冲区。
- (3) 操作系统将数据从 socket 缓冲区复制到通过网络发送的 NIC 缓冲区。

这显然是低效的，整个过程经历了四次数据复制，两次上下文切换。当然，Linux 也能够做到在数据传输的过程中，避免数据在操作系统内核态 buffer 和用户态 buffer 之间进行复制。Linux 中提供类似的系统调用函数主要有 mmap()、sendfile() 及 splice()，Kafka 采用的是 Linux 系统的函数 sendfile()，允许操作系统将数据从 Page Cache 直接发送到网络，以此来避免数据复制。

如果要想突破单机的性能，则必须拥有良好的扩展性。与其他分布式消息中间件不同的是，Kafka 可以在 Topic 下建立 Partition，Partition 才是 Kafka 的最小存储单元，这就意味着只要有足够多的分区，同一个 Topic 可以持久化到  $N$  台物理机上。另外，Kafka 提供了



重新分区的工具。

可能你会有这样的顾虑：为了保证不重复消费，同一分组的每个分区只有一个线程消费，这样是不是消费的速度太低了。事实上这是多虑了，因为取消息的过程实际上非常短，真正耗时的是业务处理的过程。有两种方式可以解决此问题：一是增加分区数；二是取消息后进行异步非阻塞处理，也就是说，把消费的线程和业务处理的线程分开。

### 3.6.5 Kafka 的数据存储结构

Kafka 的存储设计基于日志实现，非常简单。这里的日志并非我们在业务应用中打印的那种非结构化的日志，而是便于程序阅读和处理的结构化日志，它是按时间顺序排列的追加记录序列，只能在末尾添加，不能修改，以此来利用磁盘顺序写的能力。日志文件根据范围分片，每个分片都是一个片段，永远只有一个活跃文件，其余文件为只读文件，只读意味着很多操作都变得简单。如备份可以直接复制，但是读写的文件不能简单复制，因为文件随时都可能发生变化。当活跃文件到达阈值时，新建一个文件作为活跃文件，保证永远只有一个活跃文件。

Kafka 中存储的逻辑关系是 Broker>Topic>Partition>Segment，也就是 Broker 下包含多个 Topic，一个 Topic 按照范围被分为多个连续的 Partition，Partition 是 Kafka 中的最小存储单元。也就是说，一个 Partition 不能被分片后再分布到多个 Broker，只能复制多个副本分布到多个物理机，每个 Partition 对应服务器上一个物理的文件夹，每个 Partition 由多个 Segment 组成。这里不得不解释一下 Segment，既然 Partition 是最小存储单元，为什么还有 Segment 呢？实际上，Kafka 借鉴了 Bitcask 的思想，如果 Partition 是最小单元，当 Partition 不断增长的时候，就会导致单个文件特别大，这样对 Kafka 维护文件造成了非常大的影响，也不利于删除。在默认情况下，每个 Segment 包含 1GB 或 1 周的数据，哪个先到以哪个为准，写数据的过程中，如果到达上限会关闭当前文件，新建另一个文件继续写入。因此，一个 Partition 是一个文件夹，每个 Partition 下包含多个 Segment，Segment 就是 Partition 的分片。这里所说的分片是一个逻辑概念，物理上每个分片包含两种文件，一个是实际存储数据的文件，以 log 结尾，还有一个是索引文件，以 index 结尾。注意，一个 Partition 目录下的 Segment 是全集，不能放到多个文件夹里。直接看 Kafka 的 Partition 目录，大致会看到如下的文件结构。命名为起始 Message 的 Offset，Offset 是 Message 在 Partition 中的偏移量，是逻辑上的一个值，它能够唯一确定 Partition 中的一条 Message，Kafka 的消息没有明确的消息 id，因此可以认为 Offset 就是 Partition 中 Message 的 id，每个 Message 都包含以下三个属性：Offset、MessageSize、data。

```
00000000000000000000.index
```



```

00000000000000000000.log
0000000000000000434330.index
0000000000000000434330.log
0000000000000012125430.index
0000000000000012125430.log

```

举例说明一下，假设在 3 台物理机上分别部署了一个 Broker 实例，有一个名为 user 的 Topic 被分为 10 个 Partition。如果设置复制系数为 3，那么一共就有 30 个 Partition 副本。它们会被分配到这 3 个 Broker 上，在 Broker 的数据目录中会存在 user-0、user-1、user-2……user-9 文件夹，每个文件夹代表一个 Partition，其命名规则为<topic\_name>-<partition\_id>。现在有 1000 条消息发送到 Broker，消息的 Offset 是从 0 到 1000，其中 Offset 从 0 到 100 会写入一个 log 文件，Offset 从 101 到 200 会写入另一个 log 文件，并生成相应的索引，依此类推，我们认为这个 log 和 index 是一个 Segment。如果现在要查找一个消息，虽然可以利用二分查找定位到某个文件，但是在文件的哪个位置是不知道的，还是要扫描整个文件，这时候 Segment 下的 index 就派上用场了。为了提高查询效率，需要把所有的 index 文件放入内存，如果为每条消息建立索引，则会占用非常大的空间。Kafka 采用了稀疏存储的方式，每隔一定字节的数据建立一条索引，这种方法避免了索引文件占用过多的空间，缺点是没有建立索引的 Message 不能一次定位到其在数据文件中的位置，需要做一次顺序扫描，但是这次顺序扫描的范围已经很小了。

图 3-28 为论文 *Kafka: a Distributed Messaging System for Log Processing* 中的一个插图，描绘了 Kafka 中 index 和 Segment 之间的关系。

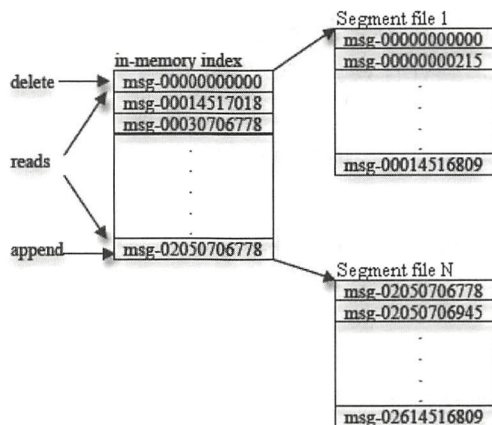


图 3-28 index 和 Segment 之间的关系



### 3.6.6 如何保证 Kafka 不丢消息

Kafka 非常灵活，高度可配置，可以适配很多场景，如果认为性能更重要，则可以通过适当降低配置可靠性来换取性能；如果认为可靠性更重要，则可以适当牺牲一些性能换取可靠性。

不过，这导致了很多研发人员认为 Kafka 容易丢消息，Kafka 是不可靠的。实际上，这并不是 Kafka 的错，而是这些人没有理解 Kafka 的设计思想，没有做针对性的配置，当配置不当的时候，数据库就会丢数据。下面从 5 个方面介绍一下 Kafka 是如何保证可靠性的。

#### 1. ACK

思考一下，假设 Service 向数据库插入一条数据，如何保证数据一定能插入数据库，不丢失数据呢？如图 3-29 所示。



图 3-29 如何保证数据不丢失

当 Service 向数据库发送 request 后，如果插入成功，数据库则向 Service 返回一个 response 表示插入成功，没错，这就是通过 ACK 机制保证消息送达。Kafka 在生产者消息投递到 Broker 中的时候，是同样的原理。

理论上，系统之间传递消息主要有如下三种方式。

- At most once: 最多一次，消息可能会丢，但绝不会重复传输。
- At least once: 最少一次，消息绝不会丢，但可能会重复传输。
- Exactly once: 有且只有一次，每条消息肯定会被传输一次且仅传输一次。

所有系统都希望能够达到有且只有一次传输的效果，但是这非常困难，代价非常高昂。Kafka 采用的是最少一次，如果是最少一次，则可能造成重复投递。ActiveMQ 可以在 Broker 端通过唯一键保证消息不重复投递，但这也仅仅是在生产者到 Broker 的阶段，当消费者从 Broker 取出消息，投递到其他服务的时候，仍然无法保证。如从 Broker 消费一条消息，并且插入数据库，仍然需要数据库一侧来保证幂等。

#### 2. 复制机制

谈到复制机制这个问题，我们必须先从消息的持久化谈起。虽然 Kafka 所有的消息都会持久化，但是并不是任何情况下都保证已经被持久化了才返回成功，一切都是基于性能的考虑。

通常 Linux 操作系统写文件有如下三种方式。



- 直接持久化到磁盘。
- 写到内核态 buffer，间隔一定时间刷新一次磁盘。
- 数据直接持久化，元数据间隔一定时间持久化。

RocketMQ 可以在配置文件中设置前两种写文件的格式，Kafka 当前版本还不可以设置，因此在单机的可靠性上 RocketMQ 优于 Kafka，但是如果设置为直接持久化到磁盘，对性能影响较大，Kafka 保证可靠性依赖的是复制机制，因为单机是容易出现故障的，恢复需要很长时间。这里需要注意的是，Broker 进程挂掉不会导致数据丢失，因为数据被缓存到了操作系统的缓存中，只有操作系统发生故障的时候，才会导致数据丢失。最好的办法就是，在其他服务器上存在一份一样的数据。

Kafka 以 Topic 为单位进行设置复制因子，以 Partition 为单位进行复制，允许一份数据复制到集群中的多个节点上。通过复制，Kafka 在 Broker 集群中的部分节点挂掉的情况下，仍然可以继续发送和接收消息。

4 个 Broker 场景下 Kafka 的复制举例，如图 3-30 所示。假如 Topic1 复制因子设定为 3，分区数为 2，当生产者向 Broker 发送消息的时候，首先计算属于哪个 Partition，如果属于 part1，则发送到 Broker0，写入 Topic1-part1-leader，Broker1 和 Broker2 下的 Topic1-part1-follower 会去拉取消息并复制到本地，一旦副本数够了，leader 就会提交。

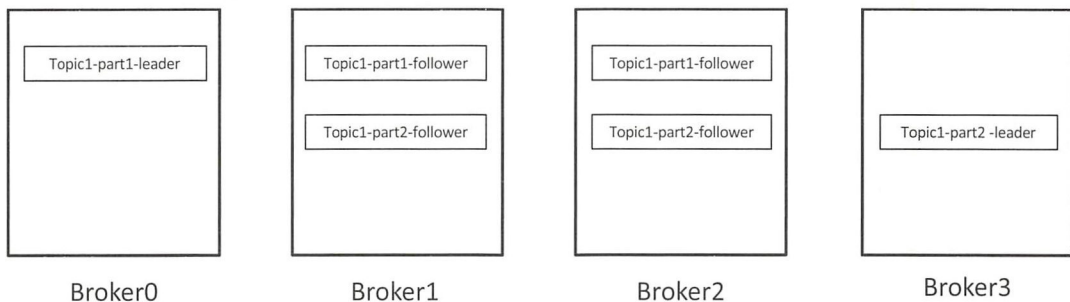


图 3-30 4 个 Broker 场景下 Kafka 的复制举例

那么，复制多少份才够呢？一旦某个 Broker 节点挂掉了，还等不等待？

Kafka 在元数据中存放了一个 ISR（In-Sync Replicas，包括 leader 和 follower。）列表，当生产者发送消息到 leader 的时候，leader 会等待消息复制到所有处于 ISR 列表中的 follower 节点都完成才会提交，一旦 follower “落后” 太多或者失效，leader 将会把它从 ISR 中删除，当 follower “追赶” 到一定范围内时，再把这个 follower 重新加入 ISR 列表。

可以在创建队列的时候通过参数指定副本数，在 `./kafka-topics.sh -topic test -create -partitions 2 -replication-factor 1 -zookeeper 10.234.10.17:2181,10.234.10.20:2181` 中



replication-factor 是副本因子。

根据 CAP 定理可知，可用性和一致性是一对矛盾的选择。如果复制因子为 3，当 ISR 列表中只有一个节点可用时，那么是给生产者返回成功还是失败呢？返回失败意味着可用性下降，如果返回成功，一旦这个节点出了问题（如磁盘坏了），数据则有丢失的可能。

在 Kafka 中，可以以 Topic 或 Broker 为单位设置最小同步副本，参数为 `min.insync.replicas`，默认值为 1，当且仅当 `request.required.acks` 参数设置为 -1 时，此参数才生效。当 ISR 中的副本数少于 `min.insync.replicas` 配置的数量时，Broker 会向生产者返回异常：

```
org.apache.kafka.common.errors.NotEnoughReplicasException: Messages are rejected since there are fewer in-sync replicas than required.
```

### 3. 消息删除机制

Broker 端删除消息有一个配置策略，默认是 7 天，如果 7 天消息还没有被消费，则有可能被删除，也就是丢消息了。我想这个时间已经够长了，如果 7 天前的消息没消费都没有被发现，那说明这个量不大或者不重要，可以把删除时间设置的长一点。

### 4. 发送消息

为了得到更好的性能，Kafka 支持在生产者一侧进行本地 buffer，也就是累积到一定的条数才发送，如果这里设置不当是会丢消息的，生产者端设置 `producer.type=async`、`sync`，默认为 `sync`。当然设置成 `async` 时会大幅提升性能，因为生产者会在本地缓冲消息，并适时批量发送。如果生产者把消息缓存到了本地，还没来得及发出去就挂掉了，那么这批消息就丢了，如果对可靠性要求高，那么这里可以设置为同步发送。

### 5. 消费消息

消费消息的时候，如果更注重可靠性，则需要显示提交 Offset，也就是当所有业务都处理完成的时候再提交 Offset，当然这可能会导致重复消费，需要提供幂等性接口。

由于可靠性是一个系统工程，需要从整体考虑，以上 5 个方面只是一些关键点，有助于提升可靠性。

## 3.6.7 Kafka 跨数据中心场景集群部署模式

某些业务场景下，我们需要多个数据中心，可能会用到多个 Kafka 集群。如果这些集群没有关系，可以部署为独立的集群，有些场景则要在多个集群之间同步数据。由于消费者是有状态的，因此这里的难点在于 Offset 的同步，一旦 Offset 不同步，则可能会导致一致性问题。

MirrorMaker 是 Apache 开源的，用于 Kafka 跨数据中心部署的镜像工具。MirrorMaker



的架构,如图 3-31 所示。MirrorMaker 包含了一组消费者,每个消费者都是一个独立的线程,不断地从原 Kafka 集群读取数据,然后通过公共的生产者发送数据到目标 Kafka 集群上,消费者默认每 60s 通知生产者发送一次数据。当收到 ACK 后,提交原 Kafka 集群的 Offset,一旦出现故障,则会重复消费 60s 的数据。由于 MirrorMaker 是无状态的,只要保证 MirrorMaker 下的消费者在同一个消费者组,就可以部署多个消费者进行扩展。

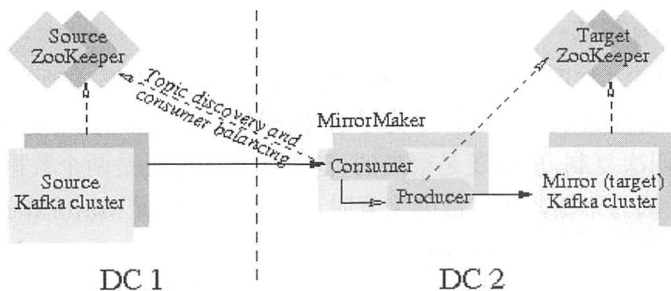


图 3-31 MirrorMaker 架构图<sup>①</sup>

MirrorMaker 的使用非常简单,可以通过以下命令执行。

```
bin/kafka-run-class.sh kafka.tools.MirrorMaker --consumer.config
sourceCluster1Consumer.config --consumer.config sourceCluster2Consumer.config
--num.streams 2 --producer.config targetClusterProducer.config --whitelist=".*"
```

基于 MirrorMaker 实现镜像有如下几种常用模式。

### 1. active/passive 模式

active/passive 模式,如图 3-32 所示。生产者只在活跃的数据中心生产数据,而备份的数据中心只能消费数据。通过 MirrorMaker 从数据中心一复制数据到数据中心二,当活跃的数据中心出现故障时,生产者切换到备份的数据中心。

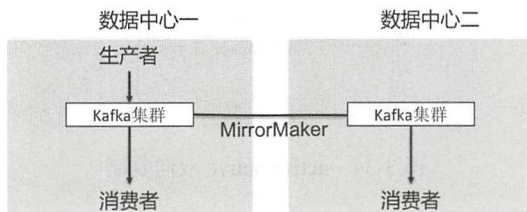


图 3-32 active/passive 模式

<sup>①</sup> 图片来自 <https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=27846330>。

这里需要注意的是，尽可能地将 MirrorMaker 部署到目标数据中心内，因为处于 MirrorMaker 中的消费者通过外部网络消费数据，要好于处于 MirrorMaker 中的生产者通过外部网络发送数据。原因是 Kafka 有持久化能力，如果消费失败，则只需要重试，不会丢失数据，消费不成功不提交 Offset 即可；但是如果是生产者发送消息失败，则需要重试，并且要考虑重试失败如何处理。

## 2. active/active 模式

active/active 模式，如图 3-33 所示。本地有两个集群，一个是本地生产者，一个是集成本地和其他数据中心的集成者，这样是为了避免循环复制。当有两个数据中心时，实际上 MirrorMaker 会做四次复制，成本还是很高的。这种模式的好处是两个数据中心都是活跃的，资源利用率更高。因为平衡，所以故障恢复时不需要重新配置 MirrorMaker。

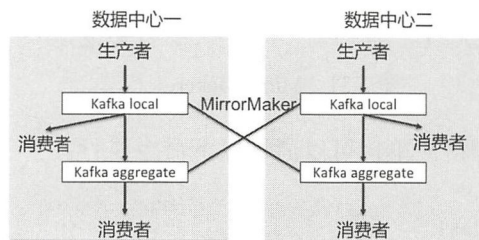


图 3-33 active/active 模式

## 3. active/active 双向复制

active/active 双向复制，如图 3-34 所示。active/active 双向复制去除集成者，通过 Topic 的名字加前缀避免循环复制，这种方式能够保证双活，部署的服务更少，架构更简单。

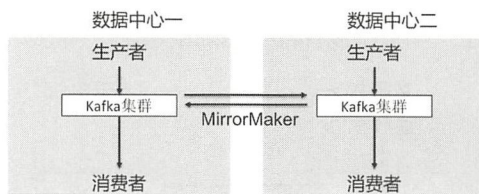


图 3-34 active/active 双向复制

## 3.7 分布式缓存服务

当系统的规模达到一定程度的时候，缓存的角色开始变得越来越重要，特别是在互联

网公司，分布式缓存被大量使用。例如在 2007 年以后，阿里巴巴分布式缓存服务器的增长速度远远超过了数据库服务器的增长速度。当然，不使用缓存也可以提升系统的吞吐量，但是所消耗的成本远远高于使用缓存的成本。在很多 SNS 类的系统架构中，缓存几乎承担了 90% 以上的数据访问流量，因此如果缓存出现问题，很容易造成系统崩溃。优秀的缓存设计可以极大地降低数据库的压力，降低系统流量高峰带来的风险。

我们从缓存的应用场景开始，逐步分析业界常用的分布式缓存 Memcached 和 Redis，最后详细描述分布式缓存的集群模式。

### 3.7.1 分布式缓存的应用场景

实际上，系统设计的各个环节都涉及缓存技术，如浏览器、CDN、代理服务器、应用服务器、数据库等处处都有缓存的影子。

按照缓存的位置分类，通常可以分为本地缓存和分布式缓存。在云化场景下，我们需要服务是可以弹性伸缩的，而服务存在状态将使弹性伸缩的难度极大提升，最好的方法就是把状态外置到数据库或者缓存中。相对于本地缓存，分布式缓存具有如下优点。

- 不需要在各个业务节点之间同步数据，如果进行数据同步，还需要考虑一致性的问题。
- 能够做到业务服务无状态，更容易扩展。
- 不需要管理数据，例如 Java，本地过多的内存会导致 FullGC 频繁发生。

当然，本地缓存也有优点，就是性能更高，不用维护额外的缓存服务。

通常，很多系统在到达一定规模时都会建立分布式缓存，但是并不是每个系统都能够利用分布式缓存，要想用好分布式缓存，首先应该明白什么样的数据是应该放进分布式缓存中的。

- 数据量比较小。因为内存比磁盘的成本要高很多，大量的数据缓存需要很高的成本。在某些情况下，需要裁剪数据，例如用户可能有 200 个属性，其中 10 个占了 99% 的访问量，其他的属性访问量并不高，这时只需要缓存 10 个属性就可以了。
- 不经常变化的数据。例如一些静态数据，比如 session 信息、配置数据，它们不经常变化，但是访问量比较大，读写比例比较高，能保证比较高的命中率。
- 计算代价比较高的数据。例如一些比较复杂的 Sql 查询，很可能关联好几张表，导致几分钟才能返回。如果不经常变化，访问量又比较高，则可以通过缓存平衡数据库的压力。
- 核心热点数据。由于内存比磁盘的成本高，因此并不是所有的数据都需要被缓存。基于用户体验的角度，放在缓存中的数据可以提升性能，间接提升用户体验。例如



SNS 类系统的核心热点是用户关系，此时可以对用户关系进行缓存。

- 同样道理，什么样的数据不应该被缓存呢？
- 变化比较快的数据。例如一个论坛的查询列表，首页可能占了访问量的 90%。如果对查询列表的首页进行缓存，未必能够达到的效果，因为评论可能会按照点赞数排序，数量的变化会导致排序的变化，结果就是命中率非常低，因为列表的排序一直在变。
- 要求强一致的数据。因为服务在写缓存的同时，需要同步更新数据库，由于网络的不确定性，在这种场景下做到强一致的代价很高。

### 3.7.2 业界常用的分布式缓存 Memcached

Memcached<sup>①</sup>是一款开源、高性能、分布式内存对象缓存系统，以简洁著称。它是一个基于内存的“键值对”存储系统，Memcached 是以 LiveJournal 旗下 Danga Interactive 公司的 Brad Fitzpatrick 为首开发的一款软件。在 Redis 以前，Memcached 是分布式缓存领域绝对的霸主，它简单且性能卓越，在大型互联网公司应用非常广泛。Facebook 是目前世界上最大规模的 Memcached 使用者，为几十亿的用户提供服务，存储着数以万亿的数据项，每秒处理几十亿的并发请求。

Memcached 具有如下特点。

- 快速，基于 libevent 的事件处理，即使服务器的连接数增加，也能发挥 O(1)的性能。可以参考 libevent 官方文档与著名的 *The C10K Problem* 一书。
- 只支持简单的 key-value 结构存储，Memcached 的高性能源于两阶段哈希（two-stage hash）结构。Memcached 就像一个可以存储很多 key-value 对的哈希表，通过 key 可以存储或查询任意的数据。
- 多线程，CPU 利用率更高。
- 内存存储方式，不能持久化，重启可能导致数据丢失。
- 不支持集群模式，需要额外开发。
- 基于 LRU 算法清理数据。

要想弄清楚 Memcached，必须先从内存模型开始研究，一般对于内存的分配有两种方式。一种是时间换空间，需要时再分配内存，也就是动态分配，这样不会浪费内存，但是会降低应用运行的效率。另外一种是用空间换时间，也就是预先分配好内存，这样做的优点是应用运行速度快，不需要等待分配内存的时间，缺点是浪费资源，显然 Memcached 需要更高的性能，因此选择空间换时间的方式。

---

① Memcached 的官网是 <https://memcached.org>。

随着 Redis 的崛起, Memcached 的关注度开始下降,但是由于 Memcached 在某些场景下内存使用效率更高,因此除了 Facebook, Memcached 仍然有大量粉丝。例如,新浪微博同时使用 Memcached 和 Redis。从 GitHub 可见其更新还比较频繁,目前还比较活跃,如图 3-35 所示。



图 3-35 Memcached 的发布频率

### 3.7.3 业界常用的分布式缓存——Redis

Redis<sup>①</sup>是一个开源的分布式内存存储系统,不仅可以用来作为分布式缓存,也可以作为分布式内存数据库、分布式消息中间件。Redis 支持的数据类型较多,包括字符串-strings、散列-hashes、列表-lists、集合-sets、序集合-sorted sets、范围查询、bitmaps、hyperloglogs 和地理空间 (geospatial) 索引半径查询。Redis 内置了复制 (replication)、Lua 脚本 (Lua scripting)、LRU 驱动事件 (LRU eviction)、事务 (transactions) 和不同级别的磁盘持久化 (persistence),并通过 Redis 哨兵 (Sentinel) 和自动分区 (Cluster) 提供高可用性。

Redis 的作者 Salvatore Sanfilippo 曾经从以下几个方面对 Redis 和 Memcached 这两种基于内存的数据存储系统进行过比较。

Redis 支持服务器端的数据操作。与 Memcached 相比,Redis 拥有更多的数据结构,支持更丰富的数据操作。通常在 Memcached 里,需要将数据拿到客户端来进行修改再 set 回去。这大大增加了网络 IO 的次数,由于数据体积增大,性能也会受到影响。而在 Redis 中,这些复杂的操作通常和一般的 GET/SET 一样高效。因此如果需要缓存支持复杂的结构和操作,那么 Redis 是不错的选择。

内存使用效率对比。如果使用简单的 key-value 存储,那么 Memcached 的内存利用率更高,而如果 Redis 采用 hash 结构来做 key-value 存储,由于其组合式的压缩,内存利用率会高于 Memcached。

性能对比。两者性能都足够高,吞吐量都接近 10 万 TPS,响应时间都为毫秒级。由于

<sup>①</sup> Redis 的官网是 <https://redis.io>。

Redis 只使用单核，而 Memcached 可以使用多核，因此平均每一个核上 Redis 在存储小数据时性能比 Memcached 性能高。而在 100k 以上的数据中，Memcached 性能要高于 Redis 的，虽然 Redis 最近也在存储大数据的性能上进行了优化，但是比起 Memcached，还是稍逊色。

Redis 和 Memcached 性能如此出色，首先它们都是基于内存操作，在内存中 Hash 查找吞吐量可以达到几百万次，因此性能瓶颈可能在网络 IO，这也就不难理解为什么 Redis 采用单线程了。单线程避免了不必要的上下文切换和锁竞争，而 Memcached 采用的是 CAS 机制。另外，两者都采用了 epoll，非阻塞 IO，利用 epoll 的多路复用特性，绝不在 IO 上浪费一点时间。

活跃度对比。Redis 更活跃一些，并且已经发布了 4.x 的稳定版。

### 3.7.4 Redis 常用的分布式缓存集群模式

单实例的 Redis 扩展性受限于单机所能达到的极限，在大规模系统中，显然是不够的。另外，单实例无法保证高可用，很容易导致缓存击穿，引起连锁反应。

Redis 在 3.x 版本之前只有 Master-Slave 模式，由于 Master-Slave 集群并没有解决写的性能瓶颈，另外 Master 和 Slave 之间是异步的数据复制，也就是说，业务应用写数据到 Master 成功之后就返回，不管有没有同步到 Slave 上。当系统对一致性要求比较高时，即使 Master 挂掉了，也不能把 Slave 提升为 Master。有一种解决方案是，利用 Redis 的 wait 指令，可以保证主从同步，但是这种 Master-Slave 结构的强同步会导致可用性下降，也就是只要有一个节点不可用就都不可用，这无疑不是我们的初衷。因此，当 Redis 官方的集群难产的时候，很多互联网公司利用 Redis 作为数据存储，在其上开发出了集群管理应用，并且十分成熟。

### 客户端模式

客户端模式架构，如图 3-36 所示。顾名思义，客户端模式也就是强调在客户端做负载均衡，不经过任何代理，直接连接 Redis 节点。客户端模式利用了 ZooKeeper 或者 Etcd 等作为注册发现服务，可以实现 Redis 节点的动态变化。

客户端模式的特点如下。

- 自动将数据集分到多个节点。例如你总共需要缓存 8GB 数据，可以分成 4 个 2GB 的实例，当一个节点不可用，只有四分之一的流量击穿缓存，不会导致崩溃。此外，4 个节点可以分布到不同的物理机，整体容量得到了提升，因为你完全可以通过一条指令分别提升每个节点的容量。



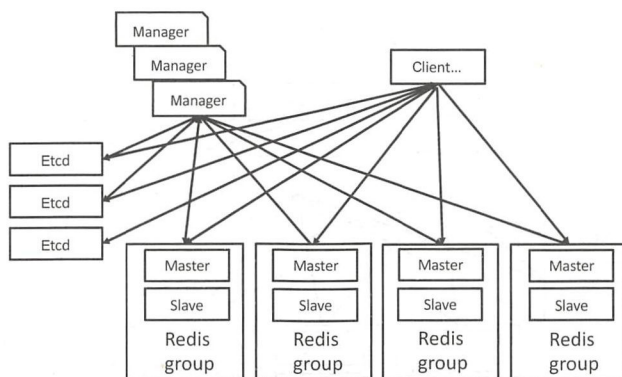


图 3-36 客户端模式架构

- 高可用。部分节点挂了仍可继续工作，如果不追求缓存可靠，可以不做持久化，不做 Master-Slave。因为持久化的两种方式都有缺陷，如果是 RDB，则有可能产生不一致；如果是 AOF，则对性能影响较大，可以根据业务场景选择。
- 通过 ZooKeeper 或 Etdcd 实现动态配置变更。
- 扩容可以采用 presharding、数据迁移两种方式。可以利用 slot 机制迁移数据，在 ZooKeeper 和 Etdcd 中存储相应的 slot 元数据，如将所有数据划分为 1024 个 slot，再以 slot 为单位分片存储到 4 个 Redis 实例上。当需要迁移数据的时候，以 slot 为单位进行迁移，要比直接根据 Redis 节点进行迁移温和得多。

客户端模式的优势是客户端直接连接 Redis，性能较高，而问题是 SDK 升级比较复杂，所幸 SDK 升级频率不高。

## 代理模式

代理负载均衡架构，如图 3-37 所示。代理模式和客户端模式类似，不同点在于代理模式把负载均衡放在代理上来做，Proxy 作为一个有效的控制点是有很多好处的，Proxy 节点对外提供标准的 Redis 协议，伪装成 Redis 节点，这样业务服务连接 Proxy 就和连接 Redis 节点一样，迁移会非常方便，开源如 Twemproxy、Codis 等都是基于代理模式实现的。

代理模式具有如下优势。

- 控制力强。
- 简单，客户端只需要连接代理服务，能力以一种服务的方式对外提供，升级方便。
- 收敛连接数。

这种方案的问题是性能有损耗，吞吐量下降可以通过扩展实例个数解决，但是响应时间下降无法解决。因此需要根据具体业务权衡，如果请求总共耗时 200 毫秒，代理只消耗

了 1 毫秒，基本可以忽略，但是如果请求总共耗时 10 毫秒，下降 1 毫秒还是需要权衡的。

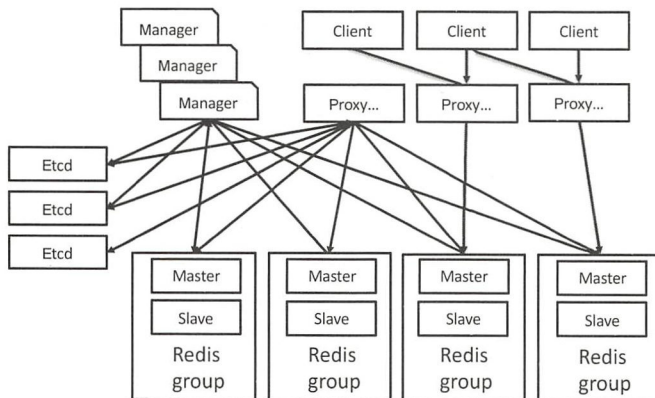


图 3-37 代理负载均衡架构

## SideCar 模式

SideCar 模式，如图 3-38 所示。SideCar 模式与代理模式很像，不同点在于 SideCar 模式的 Proxy 需要和业务服务部署在一起，这样能比代理模式得到更好的性能，并且控制力比客户端模式更好，但是它也有一个问题，业务开发人员的复杂度增加了，需要额外部署 Proxy 节点，并且多了一个调试的节点。

代理模式很容易转换到 SideCar 模式，但是采用 SideCar 模式时需要将 Proxy 节点做成无状态的。因为 Proxy 节点可能会很多，而代理模式可以将 Proxy 节点做成有状态和无状态两种形式。当做成无状态的节点时，需要利用第三方存储源数据，如 Etcd、ZooKeeper；当做成有状态的节点时，可以去除对 Etcd、ZooKeeper 的依赖。

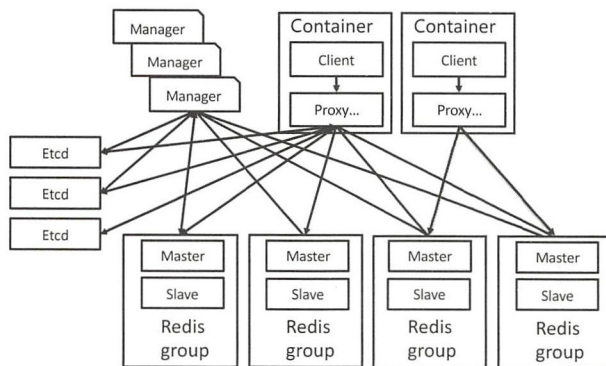


图 3-38 SideCar 模式

## 无中心模式——Redis Cluster

Redis 3.0 以后,采用了一种无中心的架构模式——Redis Cluster 架构,如图 3-39 所示。所有的 Redis 节点彼此互联(PING-PONG 机制),节点之间使用 Gossip 协议。当一个节点向另一个节点发送 PING 命令,但是目标节点未能在给定的时间内返回 PING 命令的回复时,那么发送命令的节点会将目标节点标记为 PFAIL。每次当目标节点对其他节点发送 PING 命令的时候,它都会随机广播三个它所知道的节点的信息,这些信息里面的其中一项就是说明节点是否已经被标记为 PFAIL 或者 FAIL。节点的 FAIL 是在集群中超过半数的 Master 节点检测失效时才生效。Redis Cluster 没有代理节点,客户端与 Redis 节点直接连接。此外,客户端不需要连接集群所有节点,连接集群中任何一个可用节点即可。

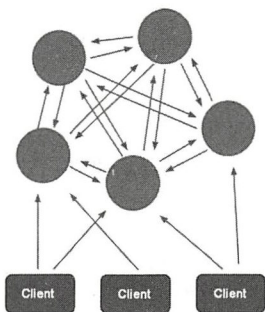


图 3-39 Redis Cluster 架构<sup>①</sup>

无中心模式的优势如下。

- 无中心架构绝对的去中心化,元数据分布在所有节点上,不会轻易丢失。
- 部署简单,存储和集群管理在一起,没有那么多依赖。
- 不经过代理,性能高。

无中心模式的问题如下。

- SDK 过重,升级麻烦。
- Redis Cluster 到目前为止还没有友好的界面管理,作者用 Ruby 实现了集群管理的大部分功能,不过还比较简单。
- 以节点为单位,分区不够小。如果你用过 Kafka、Cassandra 等架构,那么你肯定知道,分区加副本的方式可以让热点分散,热点问题在分布式缓存中尤为关键,而 Master-Slave 的方式会有一个冷的节点。当然你可以通过部署一个 Master 加一个 Slave 的方式缓解这个问题。

<sup>①</sup> 图片来自 Redis 官网。



- 不一致问题。在主从异步、重新选主过程中，Redis 集群不保证强一致性。当发生网络分区的时候，如果主服务器恰好在少数节点，实际上是有一个可以继续写入的时间窗口的，当多数节点完成重新选主，网络分区恢复时，会覆盖旧的主服务器在时间窗口内所写的数据的。
- 不够自动化。节点变动、数据迁移等操作都需要执行命令完成，和 MongoDB 比起来，还有较大差距。
- Gossip 协议导致的性能瓶颈，“总线传播的数据量 wanted 是一个变量，会随着集群实例数的增多而增多。当集群节点数不多时不会造成影响，但随着节点数变多，每次 wanted 要携带的节点信息也就随之变多，会导致对网卡的消耗越来越大。我压测时每台 Redis 服务器有 8 主 8 从共 16 个节点，4 台服务器共有 64 个节点，每个节点每秒 PING 包都要携带 6 个 gossip 包的信息（64/10），如果扩大到 10 台服务器，那么每个节点每秒 PING 包要携带 16 个 gossip 包的信息（160/10）”。<sup>①</sup>
- 绝对去中心化是很多问题的源泉。

Redis 3.x 像是一个过渡版本，而 Redis 4.x 带来了很多不一样，如今已经发布了稳定版本，值得关注。以下是 Redis 4.x 带来的新特性。

- 模块化。新的 Redis 允许开发者自定义模块扩展数据类型和函数。这个功能对于处于各种复杂业务场景的开发者绝对是一个福音。
- 部分复制。以前的版本是 Master 生成 RDB 文件，Slave 加载 RDB，很多人诟病此功能导致的全同步，后来可以通过偏移量实现部分同步，Redis 4.0 引入了 tag，解决了 Slave 提升到 Master 之后还需要进行完全同步的问题。
- 改进缓存淘汰算法<sup>②</sup>。
- 非阻塞删除，类似 Hbase 操作，先删除引用，然后真正的删除。
- 混合 RDB-AOF 格式，似乎在很多年前国内就有人用过。
- 新增内存监控命令。
- 对 NAT/Docker 的支持。
- 减少内存使用。

### 3.7.5 基于 Codis 实现 Redis 分布式缓存集群

Codis 是一个基于代理模式实现的分布式 Redis 集群解决方案。Codis Proxy 实现了 Redis

---

① 参考文章地址 <http://itindex.net/detail/54098-redis-cluster-%E7%99%BE%E4%B8%87>。

② 思路参照作者的这篇博文 <http://antirez.com/news/109>。

的协议，因此对于业务应用来说，不会感知到 Codis 和原生 Redis 的区别，业务应用像使用单机的 Redis 一样使用它即可。

截至本书定稿，Codis 最新版本为 3.x，主要包含如下组件<sup>①</sup>。

- **Codis Server:** Codis 定制版 Redis，基于 Redis-3.2.8 分支开发，增加了额外的数据结构，以支持 slot 有关的操作及数据迁移指令。
- **Codis Proxy:** 客户端连接的 Redis 代理服务实现了 Redis 协议，轻量级、无状态，对于同一个业务集群而言，可以同时部署多个 codis-proxy 实例。
- **Codis Dashboard:** 集群管理工具，支持 codis-proxy、codis-server 的添加、删除，以及数据迁移等操作。在集群状态发生改变时，codis-dashboard 维护集群下所有 codis-proxy 的状态的一致性。对于同一个业务集群而言，同一个时刻 codis-dashboard 只能有 0 个或者 1 个，所有对集群的修改都必须通过 codis-dashboard 完成。
- **Codis Admin:** 集群管理的命令行工具，可用于控制 codis-proxy、codis-dashboard 状态，以及访问外部存储。
- **Codis FE:** 集群管理界面。多个集群实例可以共享同一个前端展示页面，通过配置文件管理后端 codis-dashboard 列表，配置文件可以自动更新。
- **Storage:** 为集群状态提供外部存储。提供 Namespace 概念，不同集群的外部存储会按照不同 product name 进行组织，目前仅提供了 ZooKeeper、Etcd、Fs 三种实现，但是提供了抽象的 interface 可自行扩展。

下面简单描述一下 Codis 的架构，业务服务可以使用 redis-client 和 jodis-client 连接到 codis-proxy，如图 3-40 所示。redis-client 是指所有能够直接连接原生 Redis Server 的客户端，包括官方推荐的所有客户端，可以通过 LVS 等第三方组件实现负载均衡及故障切换。jodis-client 是 Codis 提供的客户端，使用 jodis-client 的时候可以利用 ZooKeeper 实现注册发现。codis-proxy 负责转发请求，Codis 把所有的 key 映射到 1024 个 slot (Redis Cluster 是把所有的 key 映射到了 16384 个 slot 中)中，slot 是一个逻辑概念，可以使用  $\text{crc32}(\text{key}) \% 1024$  计算 slot id。一个或多个 slot 属于一个 Server Group，1024 个 slot 被分配到所有的 Server Group 中。从理论上说，Codis 最多支持 1024 个 Redis Server。

一个 Server Group 包含一组 Redis 实例，可以只有 1 个 Master，也可以是 1 个 Master 多个 Slave。前面我们提到，由于 Redis 本身主从复制是异步的，因此如果想保证一致性，最好是不做 Master-Slave，否则当 Master 崩溃时，Slave 是不能提升为 Master 的。这一点 Codis 的作者也提到过，他们更爱一致性，但是某些对一致性要求不高的场景可以建立

---

<sup>①</sup> 组件描述来自 Codis 官方文档 [https://github.com/CodisLabs/codis/blob/release3.2/doc/tutorial\\_zh.md](https://github.com/CodisLabs/codis/blob/release3.2/doc/tutorial_zh.md)。

Master-Slave 的组。

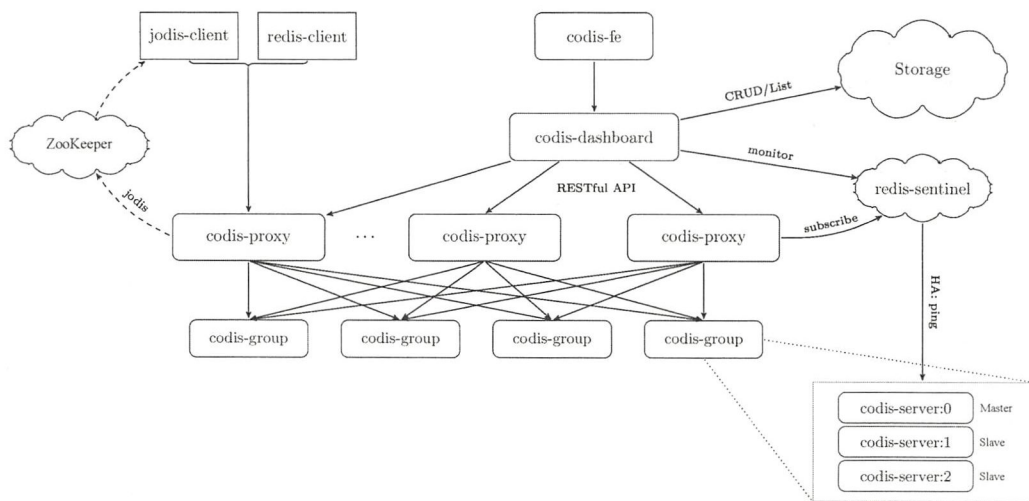


图 3-40 Codis 架构<sup>①</sup>

这些所有的元数据都会记录到 Storage 中，Storage 中主要存储了三种数据，分别是 Router、Model、Redis。Router 负责根据算法将业务应用发过来的请求转发给 Redis。Model 负责和 ZooKeeper 交互，包括 Group 的配置、Proxy 的配置、slot 的配置。Redis 模块负责与 Redis 交互。Storage 支持 ZooKeeper、Etcd、Fs 三种实现。

在正常状态下，slot 的状态为 online，存储到 Storage 中。当迁移数据的时候，修改 slot 的状态为 pre\_migrate，表示准备迁移。当所有的 Proxy 都确认以后，slot 的状态修改为 migrating，表示正在迁移，以 key 为单位不断的迁移数据。迁移完成后，将 slot 的状态修改为 online。

### 3.8 分布式任务调度服务

分布式任务调度是非常常见的一个公共服务，特别是在比较复杂的大型分布式系统中，例如电商的后端存在大量的分布式任务：按期结算、订单状态转换、退换货等，都需要按照时间、日期来启动和执行任务。

<sup>①</sup> 来自 Codis 官网 [https://github.com/CodisLabs/codis/blob/release3.2/doc/tutorial\\_zh.md](https://github.com/CodisLabs/codis/blob/release3.2/doc/tutorial_zh.md)。



一般对可用性和性能要求不高的任务，采用单点即可，例如 Linux 的 crontab，Spring 的 Quartz，但是如果对可用性的要求更高，上面的方案就不适用了。

分布式任务调度系统至少要满足以下要求。

- 不重复的执行任务。
- 不遗漏的执行任务。

举例说明一下，假设给某电商系统开发一个月末报表统计服务，当到达时间点的时候，我们要从很多张表里获取数据，并将计算结果写入某张表中。如果通过 Quartz 实现，那么服务一旦挂掉，就会导致事故。当然，还有一种方法是监控任务，当任务挂掉重新拉起时，这种方法并不知道上次执行的结果如何。如果任务比较多，而且要求在 10 分钟内统计完毕，那么怎么才能将这些任务分片呢？当任务成千上万的时候，怎么能够监控到每个任务具体执行的状态呢？下面介绍几个开源的分布式任务调度服务。

3.8.1 通过 Tbschedule 实现分布式任务调度

Tbschedule 是阿里开源的分布式任务管理服务，2012 年玄难将代码开源到 <http://code.taobao.org> 上，后期维护较少，不是很活跃，但是代码非常简单，可塑性很好。如果要求不高，则可以直接拿来用。虽然它的文档少，但是代码量也不多，可以直接通过读代码了解功能。图 3-41 展示的是 Tbschedule 的管理控制台。

图 3-42 是 Tbschedule 的架构图，基本满足了分布式任务调度的要求。ZooKeeper 有两个功能，一个是数据存储，另一个是作为分布式协调服务。管理界面直接连接 ZooKeeper 取得配置信息，并且修改配置，通过 ZooKeeper 通知任务修改配置项。



图 3-41 Tbschedule 管理控制台

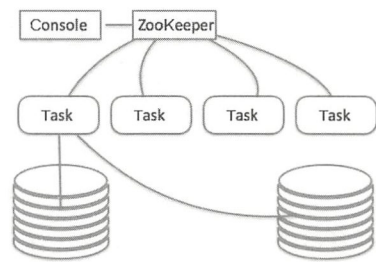


图 3-42 Tbschedule 架构图

Tbschedule 的使用过程及原理如下。

(1) 安装 ZooKeeper。

(2) 搭建 Tbschedule 控制台，直接将 console\ScheduleConsole.war 复制到自己的 Web 服务器中运行即可。启动浏览器访问 `http://localhost/index.jsp?manager=true`，通过 Console 来检查配置数据是否正确。第一次运行时，会要求你输入 ZooKeeper 的相关配置信息，如图 3-43 所示。

Zookeeper地址:	<input type="text" value="localhost:2181"/>	格式: IP地址:端口
Zookeeper超时:	<input type="text" value="3000"/>	单位毫秒
Zookeeper根目录:	<input type="text" value="/taobao-pamirs-schedule/huijin"/>	例如: /taobao-pamirs-schedule/huijin, 可以是一级目录, 也可以是多级目录。注意不同调度域间不能有父子关系, 通过切换此属性来实现多个调度域的管理。
Zookeeper用户:	<input type="text" value="ScheduleAdmin"/>	
Zookeeper密码:	<input type="password" value="password"/>	
<input type="button" value="保存"/> <a href="#">管理主页...</a>		

图 3-43 配置 ZooKeeper 相关信息

(3) 新建 Task 项目，引入 JAR 包。首先，添加 ZooKeeper 依赖。

```
<dependency>
  <groupId>org.apache.zookeeper</groupId>
  <artifactId>zookeeper</artifactId>
  <version>3.4.6</version>
</dependency>
```

其次，添加 Tbschedule 依赖。

```
<dependency>
  <groupId>com.taobao.pamirs.schedule</groupId>
  <artifactId>tbschedule</artifactId>
  <version>3.2.18</version>
</dependency>
```



## (4) 初始化配置。

```

public void afterPropertiesSet() throws Exception {
    Properties p = getProperties(configInfo);
    tbscheduleManagerFactory = new TBScheduleManagerFactory();
    tbscheduleManagerFactory.setApplicationContext(applicationContext);
    tbscheduleManagerFactory.init(p);
    tbscheduleManagerFactory.setZkConfig(convert(p));
    logger.warn("TBBPM 成功启动 schedule 调度引擎 ...");
}

```

## (5) 只需要继承 IScheduleTaskDealSingle 即可，实现代码如下所示。

```

@Component("demoTaskBean")
public class DemoTaskBean extends IScheduleTaskDealSingle<SubDetailDO> {

    /**
     * 任务数据来源，例如从一个数据库获取数据
     *
     * @param taskParameter
     * @param ownSign
     * @param taskItemNum
     * @param taskItemList
     * @param eachFetchDataNum
     * @return
     * @throws Exception
     */
    public List<SubDetailDO> selectTasks(String taskParameter, String ownSign,
                                         int taskItemNum, List<TaskItemDefine>
taskItemList,
                                         int eachFetchDataNum) throws Exception {

        try {
            Calendar calendar = Calendar.getInstance();
            calendar.setTime(new Date());
            int day = calendar.get(Calendar.DAY_OF_MONTH);
            List<SubDetailDO> details = null;

            details = subDetailDAO.selectForSchedule(
                getScopeByQueueCondition(taskItemNum, taskItemList),
                confirmTypes, DETAIL_STATUS_ONE, eachFetchDataNum);
            return details;
        } catch (Exception e) {

```



```
        log.error(e.getMessage(), e);
        throw e;
    }

}

/**
 * 处理业务的函数，例如将前面查询出来的数据插入另外一个库中
 *
 * @param subDetail
 * @param ownSign
 * @return
 * @throws Exception
 */
public boolean execute(SubDetailDO subDetail, String ownSign)
    throws Exception {
    try {
        yourProcess.process(subDetail);
        return true;
    } catch (Exception e) {
        log.error(e.getMessage(), e);
        return false;
    }
}
```

(6) 在控制台创建任务并配置策略，然后启动 Task。在控制台查看任务运行情况，进行任务配置，如图 3-44 所示。

创建新任务... 运行期信息：			
任务名称：	<input type="text"/>	任务处理的SpringBean：	<input type="text" value="demoTaskBean"/>
心跳频率(秒)：	<input type="text" value="2.0"/>	假定服务死亡间隔(秒)：	<input type="text" value="10.0"/>
线程数：	<input type="text" value="5"/>	处理模式：	<input type="text" value="SLEEP"/> SLEEP 和 NOTSLEEP
每次获取数据量：	<input type="text" value="500"/>	每次执行数量：	<input type="text" value="1"/> 只在bean实现IScheduleTaskDealMulti才生效
设有数据时休眠时长(秒)：	<input type="text" value="0.5"/>	每次处理完数据后休眠时间(秒)：	<input type="text" value="0.0"/>
执行开始时间：	<input type="text"/>		
执行结束时间：	<input type="text"/>		
任务项("，"分隔)：	<input type="text" value="0, 1, 2, 3, 4, 5, 6, 7, 8, 9"/>		
<input type="button" value="保存"/>			

图 3-44 任务配置

进行策略配置，如图 3-45 所示。

策略名称:	Tbschedule	
任务类型:	Schedule	可选类型: Schedule, Java, Bean 大小写敏感
任务名称:	DemoTask	与任务类型匹配的名称例如, 调度任务的名称, Class名称或者Bean的名称
任务参数:	逗号分隔的Key-Value	
单JVM最大线程组数量:	1	单JVM最大线程组数量, 如果是0, 则表示没有限制. 每台机器运行的线程组数量 = 总量/机器数
最大线程组数量:	10	所有服务器总共运行的最大数量
IP地址 (逗号分隔):	127.0.0.1	127.0.0.1或者localhost会在所有机器上运行
<input type="button" value="保存"/>		

图 3-45 策略配置

Tbschedule 已经满足了大多数需求, 代码写得非常优秀, 但是以下几个问题需要我们关注。

- Tbschedule 分配任务后, 为了实现高可用, 可以有多个服务实例执行任务, 如果配置为主备的结构, 备份节点是不执行任务的, 造成了资源浪费。但是, 并不是所有的情况主备都不适用, 在某些情况下, 任务处理量很少, 不需要多个节点同时工作, 只要有一个节点工作, 当一个节点崩溃时有其他节点能接替就可以了。因为取数据通常不是性能瓶颈, 瓶颈在处理数据, 多个节点的目的无非是为了高可用。如果通过 sql 取模进行分片, sql 的性能非常低, 走不了索引, 反而使性能降低了, 增大了数据库的压力。
- 实际上, 只通过简单的任务平均切分, 无法保证任务处理的均衡效果。Tbschedule 认为每台机器的配置都是一样的, 就算配置一样, 数据项不一样也容易导致其中一个节点压力特别大。需要根据机器的负载情况、程序的繁忙情况做一个加权平均来做负载。
- 执行时间修改必须在每个执行周期结束后才能生效, 它经常在调试的时候出现麻烦。这样做确实是最简单的做法, 避免了很多问题, 但是如果开发人员要配置任务每分钟执行一次, 结果把配置错写成每天执行一次, 就落入了陷阱, 等半天也看不到执行, 还以为配置错了, 重启可以解决。

### 3.8.2 通过 Elastic-Job 实现分布式任务调度

Elastic-Job 是一个开源的分布式调度中间件, 由当当开源, 在国内应用非常广泛, 且衍生出唯品会开源的分布式任务调度框架 Saturn。Elastic-Job 由 Elastic-Job-Lite 和 Elastic-Job-Cloud 两个相互独立的子项目组成。Elastic-Job 和 Tbschedule 原理类似, 都是以

ZooKeeper 作为分布式协调服务实现任务调度的。如果对任务进行分片，则可以通过对关键字进行分片的方式，如拉取某张表中的数据，对表中的主键进行分片。如果一共有两个任务处理服务，则对 2 进行求余。当其中一个任务服务不可用的时候，通过 ZooKeeper 重新进行分片。

Elastic-Job 有着很友好的文档，包括设计理念、用法指南等，Elastic-Job 也是当当从多年的电商业务实践中总结而来的，能够非常方便的应用到实践中。下面我们从官方摘录了部分文字来简单介绍一下 Elastic-Job。

Elastic-Job-Lite 为轻量级无中心化解决方案，使用 JAR 包提供分布式任务的调度和治理。Elastic-Job-Cloud 使用 Mesos+Docker 的解决方案，额外提供资源治理、应用分发及进程隔离等服务。

#### Elastic-Job-Lite

- 分布式调度协调。
- 弹性扩容、缩容。
- 失效转移。
- 错过执行作业重触发。
- 作业分片一致性，保证同一分片在分布式环境中仅一个执行实例。
- 自诊断并修复分布式不稳定造成的问题。
- 支持并行调度。
- 支持作业生命周期操作。
- 丰富的作业类型。
- Spring 整合及命名空间提供。
- 运维平台。

#### Elastic-Job-Cloud

- 应用自动分发。
- 基于 Fenzo 的弹性资源分配。
- 分布式调度协调。
- 弹性扩容、缩容。
- 失效转移。
- 错过执行作业重触发。
- 作业分片一致性，保证同一分片在分布式环境中仅一个执行实例。
- 支持并行调度。
- 支持作业生命周期操作。



- 丰富的作业类型。
- Spring 整合。
- 运维平台。
- 基于 Docker 的进程隔离（TBD）。

由于 Elastic-Job 的官方文档比较详细，本书不再介绍相关内容，详细内容可参考官网<sup>①</sup>。通过 Elastic-Job-Lite 创建一个简单任务的过程大致如下。

首先，通过 Maven 引入相关 JAR 包。

```
<dependency>
  <groupId>io.elasticjob</groupId>
  <artifactId>elastic-job-lite-core</artifactId>
  <version>${latest.release.version}</version>
</dependency>
<dependency>
  <groupId>io.elasticjob</groupId>
  <artifactId>elastic-job-lite-spring</artifactId>
  <version>${latest.release.version}</version>
</dependency>
```

然后，开发一个 SimpleJob 类型的服务，只需要实现 SimpleJob 接口即可。

```
public class MyJob implements SimpleJob {

    @Override
    public void execute(ShardingContext context) {
        switch (context.getShardingItem()) {
            case 0:
                // do something by sharding item 0
                break;
            case 1:
                // do something by sharding item 1
                break;
            case 2:
                // do something by sharding item 2
                break;
            // case n: ...
        }
    }
}
```

<sup>①</sup> Elastic-Job 官网的网址 [http://elasticjob.io/index\\_zh.html](http://elasticjob.io/index_zh.html)。

```
}  
}
```

最后，通过 Spring 配置文件定义任务。配置 ZooKeeper 地址获取任务，通过 crontab 表达式设置任务启动时间，示例表示每隔 12 小时执行一次。

```
<reg:zookeeper id="regCenter" server-lists="yourhost:2181" namespace="dd-job"  
base-sleep-time-milliseconds="1000" max-sleep-time-milliseconds="3000" max-retries="3"  
</>  
<job:simple id="oneOffElasticJob" class="xxx.MyJob" registry-center-ref="regCenter"  
cron="0 */12 * * * *" sharding-total-count="3" sharding-item-parameters="0=A,1=B,2=C"  
</>
```

## 3.9 如何生成分布式 ID

通常情况我们会采用数据库自增的方式生成 ID，但是当业务比较复杂，数据量比较大，需要水平分表的时候，就不得不自己建立分布式 ID 服务了。

从安全性方面考虑，一个纯粹的自增 ID 是可以被利用的，所以 ID 一般不是从 1 开始自增的。为了水平分表时数据查询方便，一般可能会在 ID 上加上语义，如设计电商中订单 ID 时，将订单 ID 与用户 ID 进行拼接处理，保持订单 ID 和用户 ID 的末尾数字相同。当按照订单 ID 水平分表时，以用户 ID 查询就可以落到一个分片上。

从性能角度考虑，如果一个主键非常长，则可以计算一下将消耗多少资源。例如页面上一个列表的<A href=<http://xx.aa.com/id=32> 位 ID>可能有几十条，网络资源、缓存、数据库的消耗将是非常可观的。

下面为大家介绍几种常用方案，可以根据具体业务场景来选择。

### 3.9.1 UUID

UUID<sup>①</sup>是指在一台机器上生成的数字，它保证在同一时空中的所有机器都是唯一的。通常平台会提供生成 UUID 的 API，按照开放软件基金会（OSF）制定的标准计算，用到了以太网卡地址、纳秒级时间、芯片 ID 码和许多可能的数字。

UUID 由以下几部分的组成。

---

① 引自百度百科，“UUID 是通用唯一识别码（Universally Unique Identifier）的缩写，是一种软件建构的标准，亦为开放软件基金会组织在分布式计算环境领域的一部分。其目的是让分布式系统中的所有元素都能有唯一的辨识信息，而不需要通过中央控制端来做辨识信息的指定。”

- 当前日期和时间。
- 时钟序列。
- 全局唯一的 IEEE 机器识别号，如果有网卡，则从网卡 MAC 地址获得，没有网卡则以其他方式获得。

有 4 种不同的基本 UUID 类型：基于时间的 UUID、DCE 安全的 UUID、基于名称的 UUID 和随机生成的 UUID。

在 Java 中使用 UUID 非常方便，可通过如下代码实现。

```
UUID uuid = UUID.randomUUID();
```

运行结果为 35b3dda6-f481-410a-b393-9adc91703c68，可以看到这是一个字符串，并不是数字。

UUID 具有如下优点。

- 使用方便，很容易实现。
- 性能很高。

UUID 具有如下缺点。

- 没有顺序，不能保证单调递增。
- 太长，总长 32 位，无论是存储还是传输，缺点都比较明显。

### 3.9.2 SnowFlake

Twitter 的 SnowFlake<sup>①</sup>是一个非常优秀的 ID 生成方案，实现也非常简单，8Byte 是一个 Long，8Byte 等于 64bit，核心代码就是毫秒级时间 41 位+10 位机器 ID+毫秒内序列 12 位，也可以调整机器位数和毫秒内序列位数比例。实际上从这个项目可以衍生出很多实现方式。SnowFlake 是用 Scala 实现的。

SnowFlaker 的优点如下。

- 比 UUID 短，一般为 9~17 位。
- 生成的 ID 是数字，可以做到单调递增。注意，由于无法统一分布式环境中每台服务器的时钟，它只能做到在单台机器单调递增，无法做到全局递增。
- 性能非常出色，吞吐量达到几十万 TPS。

SnowFlaker 是 UUID 的一种替代方案，如果能用 SnowFlaker 的方式，绝对不用 UUID。注意，SnowFlake 算法一旦确定，时间序列的位数就确定了，超出时间范围将会溢出。

<sup>①</sup> <https://github.com/twitter/snowflake>。



### 3.9.3 Ticket Server

Ticket Server 是 Flickr 采用的一种分布式 ID 生成方案，利用 MySQL 自增长 ID 实现。它的设计思路是利用数据库中 `auto_increment` 的特性和 MySQL 特有的 `REPLACE INTO` 命令来实现，可以利用多台 MySQL 实现高扩展性和高可用性，过程如下。

首先，启动两个 MySQL 实例，分别进行配置。

节点一进行如下设置。

```
auto_increment_increment=2;  
auto_increment_offset=1;
```

节点二进行如下设置。

```
auto_increment_increment=2;  
auto_increment_offset=2;
```

注意，这两个配置最好放在配置文件中，否则 MySQL 服务重启将丢失设置。如果直接通过 `set` 来设置，那么已经建立的连接不会改变变量的值，特别是使用连接池的时候。

其次，分别在两个库里建表。

```
CREATE TABLE `tickets1` (  
  `id` bigint(20) unsigned NOT NULL auto_increment,  
  `stub` char(1) NOT NULL default '',  
  PRIMARY KEY (`id`),  
  UNIQUE KEY `stub` (`stub`)  
) ENGINE=MyISAM
```

注意，上面的代码建的是 MyISAM 表，它是表级锁，能保证所有 `REPLACE` 的原子性。不断地通过 `REPLACE` 促使 ID 自增，这样表中只有一条记录。

```
REPLACE INTO tickets1 (stub) VALUES ('a')REPLACE INTO tickets1 (stub) VALUES ('a')
```

在同一个连接内，通过 `last insert id` 获取自增的 ID 值。对 MyISAM 性能影响比较大的参数是 `key_buffer_size`，可以适当调整。

Ticket Server 的优点如下。

- 对于数据量不是特别大的应用，长度最小，从零开始。
- 如果已经有应用是基于 Oracle 中的 `sequence` 或者 MySQL 的自增 ID，那么采用此方案非常容易迁移，而且兼容性好。

需要注意的是，这种方案是没有绝对顺序的，只能有一个近似顺序，有可能在某个实例的状态下跑得更快。

### 3.9.4 小结

首先要保证 ID 是全局唯一的，否则 ID 就失去了意义。另外，由于无法统一分布式环境中每台服务器的时钟，无法做到全局递增，因此 A 服务可能取 ID 早，但是入库晚，B 服务可能取 ID 晚，但是入库早，这就会造成后入库的 ID 比先入库的 ID 小，这是和单机的递增 ID 有区别的。因此，分布式 ID 无法保证全局的顺序性，只要有大概的顺序，对于调试、简单识别而言足够了。

如果在高并发、大数据量的情况下，建议采用 SnowFake 方案，它的性能非常突出。而对于需要兼容老业务的应用，特别是拿到 ID 要进行拼装以表示某种意义，并发又不那么高的情况下，可以使用 Ticket Server 方案，两台物理机也能轻松到达几万 TPS，它的响应时间在毫秒级。

# 4

## 第4章 可用性设计

提高可用性需要从全局考虑，这里不单指技术，研发流程和团队文化也会影响可用性。有人问是不是做好某些关键点就可以做到高可用了？答案是否定的，不是做好几个关键点就可以了，只能说这些方法、手段对提升可用性有好处。提升可用性是一个过程，通常可用性需要长时间的积累，很难在短期内实现高标准，当然，这里也存在运气的成分。下面详细介绍可用性架构设计的关键点。

### 4.1 综述

#### 4.1.1 可用性和可靠性的关系

首先让我们了解一下可用性和可靠性的关系，这两个概念虽然比较接近，但是还是有区别的。

- 可用性（Availability）是关于系统可以被使用的时间的描述，以丢失的时间为驱动（Be Driven by Lost Time）。
- 可靠性（Reliability）是关于系统无失效时间间隔的描述，以发生的失效个数为驱动（Be Driven by Number of Failure）。

两者都可以用百分比的形式来表示。

- 可用性公式： $A = \text{Uptime} / (\text{Uptime} + \text{Downtime})$ 。其中，Uptime 是可用时间，Downtime 是不可用时间。



- 可靠性公式： $A = MTBF / (MTBF + MTTR)$ 。其中，MTBF 的全称是 Mean Time Between Failure，即平均无故障工作时间，指上一次故障恢复后开始正常运行到这次故障的时间平均值。MTTR 的全称是 Mean Time To Repair，即平均故障修复时间，是指从出现故障到完全恢复的这段时间。

4.1.2 可用性的衡量标准

可用性通常以  $N$  个 9 的方式来量化衡量。用于衡量可用性的指标一般有两个，一个是对外承诺的可用性，一个是生产环境的测量值。例如，AWS 的 ECS 服务对外承诺的可用性是 99.95%，当低于这个值的时候，可以得到相应的赔偿。

要做到更高的可用性，所要付出的代价也更高，而且即使你用了很多技术，也不一定能运用好。此外，发布的次数、访问量都会对可用性提出挑战。可用性等级，如表 4-1 所示。

表 4-1 可用性等级

可用性等级	通俗叫法	可用性百分比	年度停机时间	可能会用到的技术
基本可用	2 个 9	99%	87.6 小时	简单的负载均衡
较高可用	3 个 9	99.9%	8.8 小时	灰度发布、自动化发布、自动化测试、快速回滚
高级可用	4 个 9	99.99%	53 分钟	微服务、相关中间件自动扩展（数据库自动扩展、缓存自动扩展）、容错、监控、弹性伸缩
极高可用	5 个 9	99.999%	5 分钟	异地多活、智能运维

4.1.3 什么降低了可用性

以下几点是导致可用性下降的常见原因。

- 发布。当应用需要升级的时候，为了得到更好的用户体验，应用不能中断，如果需要迁移数据，会导致整个流程相当复杂。为了降低复杂度和开发成本，我们通常会暂时中断服务。
- 故障。发生故障的时候，系统可用性会受到影响，例如出现内存溢出，可能导致整个服务不可用。当然并不是所有的故障都会导致不可用，例如一个商品详情页中的推荐购买不显示了，实际上并没有影响可用性，但是如果价格不可见了，那么用户不能提交订单，这就影响了购买商品的可用性。
- 压力。很多宕机都是因为突发的事件导致的，例如某某明星在微博发布“介绍一下我的女朋友”信息，预期之外的访问压力会造成系统宕机，而预留太多冗余资源又比较浪费。所以部署到公有云或者基于公有云做混合云是一种既可以应对超预期的流量又省钱的办法。

- 外部强依赖。如果外部依赖的服务发生故障，则会导致调用异常，进而导致系统的不可用。外部依赖的服务越多，做到高可用的挑战就会越大。

要实现高可用，需要在设计阶段考虑如下几个比较重要的方法。

- 20/10/5，设计系统的时候，以实际流量的 20 倍来设计；开发系统的时候，以实际流量的 10 倍来开发系统；发布系统的时候，以实际流量的 5 倍来部署。这只是一个通用的原则，可以根据实际情况来确定，不需要严格按照倍数来执行。
- Design for failure，预测可能发生的问题，做好预案。例如当流量高峰的时候如何限流、伸缩、隔离故障节点。

下面，我们再介绍几个对提升可用性比较重要的方法。

## 4.2 逐步切换

现在的你，还在等到凌晨才敢上线吗？不区分业务高峰和低峰上线服务，不是一种胆量，也不是运气，而是有一套成熟的机制做后盾。

### 4.2.1 影子测试

影子测试是一种常用的在生产环境中通过流量复制、回放和比对的测试方法。对比验证，如图 4-1 所示。先同步新老数据库内的数据，在不影响老服务的情况下，在负载均衡的位置记录请求日志，通过日志回放服务向新服务发送请求，新服务正常处理业务逻辑后入库，对比验证服务、对比老数据库和新数据库之间的差异。如果所有比对都正确，则说明新服务和老服务逻辑上是等价的。

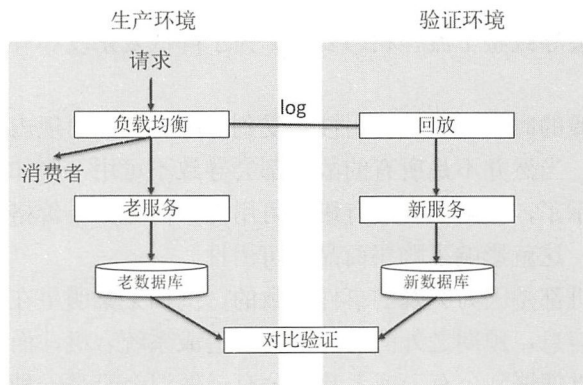


图 4-1 对比验证

以上是基于日志实现的，还可以使用 TCPCopy 实现，直接复制生产环境上的请求实现验证。当然如果服务间存在 MQ，直接添加消费者进行消费也比较方便。

影子测试可以直接基于生产环境的流量进行测试，并且对生产环境无影响，解决了线下测试“不准确”的问题，因此目前被很多大型互联网公司所采用。

这种测试方法需要额外冗余一定的资源，依赖越多就会越复杂。

## 4.2.2 蓝绿部署

蓝绿部署是一种以可预测的方式发布应用的技术，目的是减少发布过程中服务停止的时间。简单来说，在生产环境中，除了正在运行的环境（蓝色环境），需要额外冗余另外一份相同的环境（绿色环境，如图 4-2 中的 v1 所示）。在蓝色环境运行当前生产环境中的应用，如图 4-2 中的 v2 所示。

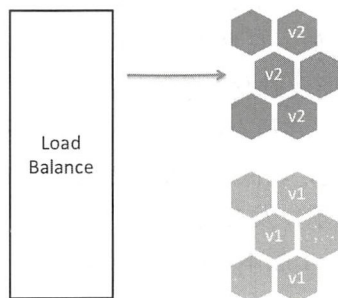


图 4-2 蓝绿部署（一）

如果要升级到 v3，则在绿环境部署 v3 版本，即部署新版本。测试通过后，可以把负载均衡器/反向代理/路由指向绿色环境，一旦发生故障需要回退时，只需要切换到蓝色环境即可，如图 4-3 所示。

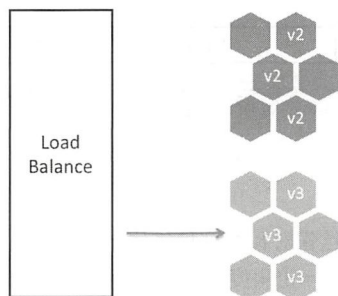


图 4-3 蓝绿部署（二）



蓝绿部署听起来很棒，但要注意如下细节。

- 最好有自动化的基础设施作为支撑。
- 全面的监控，发生故障马上发送告警。
- 两套环境隔离问题，有相互影响的风险。
- 难点是当数据结构发生变化时，如何同步数据，发生故障时，如何回滚。
- 切换时需要优雅的终止机制，禁止直接 kill 进程。

采用蓝绿部署回滚速度会比较快，只要切流量就能实现，但是比较浪费资源，需要有一套独立的闲置资源。

### 4.2.3 灰度发布/金丝雀发布

金丝雀发布和灰度发布非常像，本节把两者放在一起描述。

小说《鬼吹灯》里有一个场景，盗墓之前需要先点一根蜡烛，如果蜡烛熄灭，就要停止一切活动，另外一个方法就是放一只鸟进去探测空气质量。实际上，古代的矿井工人也经常利用金丝雀探测瓦斯<sup>①</sup>，以便及时发现危险情况。

灰度发布利用了同样的原理，在原有版本可用的情况下，同时部署一个新版本作为“金丝雀”，测试新版本的性能和表现，以保障在整体系统稳定的情况下，尽早发现问题并解决问题。金丝雀发布，如图 4-4 所示，发布过程如下。

- (1) 假设生产环境运行的是 v1 版本，从负载均衡列表中摘掉一个节点，作为“金丝雀”服务器。
- (2) 在“金丝雀”服务器上部署 v2 版本。
- (3) 进行自动化测试。
- (4) 将 v2 版本节点添加到负载均衡列表中。
- (5) 如果发生故障，则马上进行回滚。
- (6) 如果没有问题，则逐步升级剩余的其他节点。

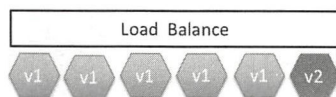


图 4-4 金丝雀发布

① 17 世纪，英国矿井工人发现，金丝雀对瓦斯这种气体十分敏感。空气中哪怕有极其微量的瓦斯，金丝雀也会停止歌唱；当瓦斯含量超过一定限度时，虽然鲁钝的人类毫无察觉，但是金丝雀却早已中毒死亡。在采矿设备相对简陋的条件下，工人每次下井都会带上一只金丝雀作为瓦斯检测指标，以便在危险状况下紧急撤离。



灰度发布的意义如下。

- 减小故障的波及范围。因为变更更容易导致故障的发生，而 90% 的故障可以在早期发现。通过逐步切换的方式，相当于扩大了测试范围，以发现隐藏的故障点，从而提升系统的可用性。
- 尽早得到用户的反馈。详尽的分析远远比不上快速试错得到的结果有说服力，我们不能等产品 100% 完美才发布。以发布和功能叠加为目的的产品演进是失败的，应该以数据驱动、用户体验为中心进行产品演进。例如某注册流程，有可能因为在注册的过程中增加了住址为必填项导致转化率大幅度降低，通过大数据分析反馈得到结果，此时应该将这个属性转移到当用户必须使用这个数据的时候再填写，如电商中只有下订单的时候可能才会用到，浏览商品只需要城市即可。

以上是简单的原理介绍，一般互联网公司都会有独立的灰度发布引擎，如图 4-5 所示。由运维人员设置规则，导入友好用户，当外部请求进入的时候，在负载均衡的位置调用灰度发布引擎的规则解析，判断请求是否进入新版本，然后进行转发。这个灰度的过程一般是：内部员工>外部 1%友好用户>5%友好用户>10%友好用户>全网发布。

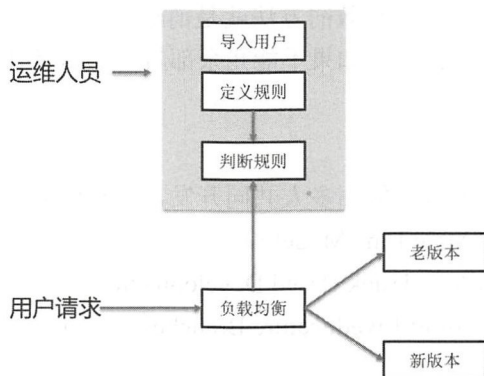


图 4-5 灰度发布引擎架构图

灰度发布可以根据任意维度进行切分，但是通常根据用户相关的属性进行切分，如用户所在的区域、用户的级别等。另外，灰度发布通常需要有一个过程，我们需要引入一套自动化的流程进行发布，需要通过快速的发布回滚机制和全面的监控报警来保证高可用。

### 4.3 容错设计

如果说错误是不可避免或者难以避免的，那么我们应该换一个思路，保证错误发生时，



我们可以从容应对。

### 4.3.1 消除单点

有一个故事，当飞行员学习飞行时，要飞到足够的高度，给自己预留足够的操作失误的时间，那么要飞多高呢？飞的高度至少要允许连续出现两次失误，这就是著名的“两次失误高度”。

当你希望系统有更高的可用性的时候，首先要做的就是消除单点，通过负载均衡分配流量，部署多个业务服务，存多份数据。

当然，节点数越多，可用性就越高，因为多个节点同时发生故障的概率更低，但是冗余太多的节点也会浪费更多的资源，因此需要根据实际情况做出选择。

那么，为什么一般都要求节点数为  $n+2$  个而不是  $n+1$  个呢？有一种说法是，当你进行升级的时候，此时要升级一个节点，如果剩下两个节点正常提供服务，那么还可以允许一个节点失效。如果一共只有两个节点，升级一个节点，就只剩一个节点对外提供服务，那么一旦发生意外，就会比较惨。这无形中给了部署人员压力，压力之下更容易出错。

要提升可用性，最简单、最有效的方法就是消除单点，部署多个节点。检查生产环境上所有的服务是否都为单点部署，如果不能冗余部署，那么是什么原因导致的。

### 4.3.2 特性开关

在开发阶段，同一个服务可能由多人共同开发，常见的分支模型如下。

- 稳定 Trunk 模型（Main Line Model）。
- 基于 Trunk 开发模型（Trunk Based Development）。
- 短生命周期分支（Short Lived Feature Branches Model）。
- 级联模型（Cascade）。

大多数互联网公司都是基于主干开发的，包括 Google 和 Facebook。主干开发最大的好处是避免了合并分支时的麻烦。但是由于微服务架构场景下，要求频繁交付，下面两种情况是经常遇到的。

- 在多人同时协作开发的时候，可能 A 开发的特性 M 已经完成，B 开发的特性 N 还没完成，为了不影响 M 上线，我们可以采用特性开关关闭 N。
- 当 M、N 两个特性都要上线时，发现 A 出现 bug，是否要全部回退？能否通过开关关掉 A？如果是移动版或者是桌面客户端，升级版本是一个非常烦琐的过程，需要花费较长时间。

简单的特性开关可以通过配置文件或者程序中的一个静态变量来实现。但是，当部署





实例比较多的时候，一个个修改配置是比较烦琐的，而且容易出错。最好是建立一个通用的特性开关服务，例如阿里的 switch。图 4-6 简单描述了通过特性开关如何控制特性的状态。

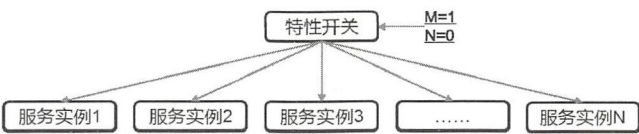


图 4-6 特性开关

服务实例可以每 5 秒轮询一次特性开关服务，或者特性开关服务出现变化时通知服务实例。特性开关服务的管理控制台可以实时查看到每个服务实例开关的当前状态。

另外，持续集成不只是靠特性开关这么简单，还要在代码提交阶段遵循很多规则，提升自动化测试覆盖率，做好 Code Review 等。

4.3.3 服务分级

在微服务架构中，服务之间通过契约化的接口进行调用，服务数量越多，依赖关系就会越复杂。服务调用关系，如图 4-7 所示。用户请求商品详情页，可能要调用很多服务才能完成请求。对于用户来说，商品信息、库存、价格非常重要，而相似推荐即使没有也不会造成太大的影响。如果因为相似推荐导致整个页面不可用，那就非常糟糕了。所以，当服务数量较多的时候，我们应该梳理出业务的核心流程，找到核心的服务是哪些，根据这些给服务分级。

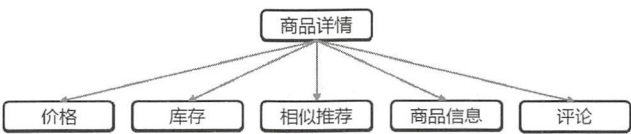


图 4-7 服务调用关系

服务分级实际上就是服务的标签，表示服务的关键程度。通常我们会把服务分成 4 个级别，当然，根据业务复杂度，也可以分更多的级别。

表 4-2 以电商为例描述了如何对服务进行分级。

表 4-2 电商领域的服务分级

服务级别	划分依据	示 例
1 级服务	核心业务流程一旦发生故障，将直接导致业务遭受重大损失	下单服务、价格服务、库存服务



续表

服务级别	划分依据	示 例
2 级服务	用户体验受到严重影响。一旦发生故障，关键业务还可以用，但用的过程中用户体验受到严重影响	评论服务、搜索服务、物流服务
3 级服务	用户体验受到轻微影响。一旦发生故障，正常业务流程不受太大影响，一些不常用的功能不可用	推荐服务、个人信息、积分服务
4 级服务	多为管理维护服务，用户不会受到影响，用户不会直接访问	报表统计、舆情分析

### 4.3.4 降级设计

当高出预期的流量来临时，无法做到快速扩展，也就是加机器起到的效果不理想，我们只能通过流控和降级来解决问题。阿里巴巴在“双 11”活动期间经常会关闭一些无关紧要的功能，或者对它们进行降级。例如确认收货，买家按“确认收货”按钮后，钱会实时转入卖家账户，买家获得积分；如果买家不按“确认收货”按钮，等到一定时间后，也会有定时任务触发相关操作。因此，阿里在双十一流量高峰的时候会隐藏这个按钮，达到降级的效果。

降级是指为了保障核心功能，利用目前有限的资源，通过开关手段暂时关闭非核心服务。暂时关闭意味着给用户体验带来一些影响，是有损操作。

那么，通常会采用哪些降级方式呢？

- 关闭某个功能，页面显示不全或不能点击某个按钮。
- 请求短路，直接返回缓存结果。
- 简化流程，放弃某个操作，如给用户发注册成功短信。
- 延迟执行，停止定时任务，如某些结算。

降级的前提条件是要对服务进行分级，需要在设计阶段明确降级的条件，是吞吐量太大了，还是响应时间太长了，或者是由于依赖的某个服务不可用了？1 级服务是核心服务，是绝对不能降级的，而 4 级服务可以立即进行降级。根据情况决定是否对 3 级服务进行降级。

降级的方法如下。

- 页面加开关，通过 JS 控制功能是否隐藏。
- 关闭低级别服务前端页面。例如一些运营系统，为了统计、反馈，这些运营系统可能会访问某个核心服务，可以直接关闭前端页面应用。
- 关闭定时任务。有一些非核心的，每天需要运行的服务可以关闭，例如电商中结算系统每天凌晨会统计当日数据，这个服务可以暂时关闭。
- 预先定义降级逻辑。在配置中心定义一个变量，预先定义好变量的含义，例如变量



的值为 3，则要求所有的 3 级以下的服务都不调用，可以结合微服务框架，通过框架的隐含参数来实现。在紧急情况下，可以启用此按钮。

- 降低精确度。例如在电商中，库存可以显示为有货或者无货，而不是具体的数量。价格可以不那么及时更新，当提交订单的时候再计算最新值，毕竟浏览的人多，下单的人少。

总之，降级是不得已而为之的，至少比大面积宕机的用户体验好得多。降级需要事先测试，并且进行演练。否则当问题真正来临的时候，这个按钮可能导致更大的故障。另外，最好给用户友好的提示，如“系统繁忙，码农正在努力给您下单，稍后会给您发送成功消息”。

### 4.3.5 超时重试

服务调用，如图 4-8 所示。生产者调用消费者，请求的状态分为成功、失败、超时三种。超时是所有状态中最难以处理的，因为超时通常是生产者发起的，生产者并不知道这次请求的结果是什么，有可能数据正在被消费者处理，也有可能失败了。如果频繁发起重试，则可能会加重消费者的负担。

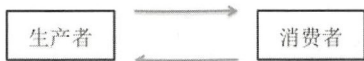


图 4-8 服务调用

为了避免重试导致更严重的事故，在制定超时重试的策略时，需要考虑的参数如下。

- 超时时间。
- 重试的总次数。
- 重试的间隔时间。
- 重试间隔时间的衰减度。

以下两个案例是我在工作中遇到的。

#### 案例一

在图 4-9 中，某运维监控系统 M 监控某服务 S，M 每隔 2 秒发送一次心跳到 S，超时时间设为 1 秒，重试三次后如果还失败，则重启服务 S。当服务 S 依赖的其他服务发生问题，导致三次监控系统的重试全部失败时，监控系统重启服务 S，上游服务再全部重试一遍，整个系统发生级联故障。这种重启的方式治标不治本，设置重启参数的时候应该参考服务 S 依赖服务的执行时间和超时时间设置。







图 4-9 案例一

## 案例二

某服务 x 非常耗时，98%的请求执行时间要 1 秒以内，1%的请求执行时间要 10 秒以上，此时当服务 y 调用服务 x 时，为了降低错误率，服务 y 设置超时时间为 10 秒。由于服务 x 个别请求的超长执行时间导致服务线程池耗尽，影响正常业务执行。在这种场景下，不应该为了保证个别请求而降低整体的可用性，应该设置更低的超时时间。

超时时间的设置要非常谨慎，通常没有一个通用值，需要根据环境不断调整。我认为，机器学习可以很好地解决这个问题，通过 AI 对一些需要动态改变的参数进行预测和动态配置。

下面介绍几种超时重试的模式。

### 1. 简单重试模式——try-catch-redo

在正常业务逻辑处理的过程中，一种情况是返回结果含错误码，可以在返回后直接判断，直接进行重试；另一种是出现异常，捕获远程调用方法抛出的异常，在 catch 中进行重试处理，并且可以增加适当的休眠时间。

代码示例如下。

```
/**
 * try-catch-redo 简单重试模式
 */
public void retrySimple() {
    Map<String, Object> paramMap = Maps.newHashMap();
    paramMap.put("x", "1");
    paramMap.put("y", "2");
    int result = 0;
    try {
        result = invoke(paramMap);
        if (result != 200) {
            Thread.sleep(1000);
        }
    }
```



```

        invoke(paramMap); //第一次重试
    }
} catch (Exception e) {
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e1) {
        e1.printStackTrace();
    }
    invoke(paramMap); //第二次重试
}
}
/**
 * 模拟业务调用方法
 * @param paramMap
 * @return
 */
private int invoke(Map<String, Object> paramMap) {
    log.info("invoke-----");
    return 400; //模拟异常
}

```

## 2. 策略重试模式——try-catch-redo-retry strategy

上述方案还是有可能重试无效，为了解决这个问题，可以尝试增加重试次数 `retrycount`、重试间隔周期 `intervalTimey`，以及重试间隔周期衰减的实际 `weakTime`，从而增加重试有效性。

```

/**
 * try-catch-redo-retry strategy 策略重试模式
 */
public void retryStrategy() {
    Map<String, Object> paramMap = Maps.newHashMap();
    paramMap.put("x", "1");
    paramMap.put("y", "2");
    int result = 0;
    try {
        result = invoke(paramMap);
        if (result != 200) {
            invokeRetry(paramMap, 1000L, 100L, 10); //延迟多次重试
        }
    } catch (Exception e) {
        try {
            invokeRetry(paramMap, 1000L, 100L, 10); //延迟多次重试
        } catch (InterruptedException e1) {
            e1.printStackTrace();
        }
    }
}

```



```

    }
}

/**
 * 递归重试调用
 * @param paramMap
 * @param intervalTime
 * @param retryCount
 *
 * @return
 * @throws InterruptedException
 */
private int invokeRetry(Map<String, Object> paramMap, long intervalTime, long weakTime,
int retryCount) throws InterruptedException {
    log.info("invokeRetry retryCount="+retryCount);
    Thread.sleep(intervalTime);
    int result = invoke(paramMap);
    if (result!=200&&retryCount>1){
        invokeRetry(paramMap, intervalTime+weakTime, weakTime, retryCount-1); //递归重试
    }else if (retryCount==1){
        log.info("retry completed");
    }else {
        log.info("success");
        return 200;
    }
    return result;
}

/**
 * 模拟业务调用方法
 * @param paramMap
 * @return
 */
private int invoke(Map<String, Object> paramMap) {
    log.info("invoke-----");
    return 400; //模拟异常
}
}

```

运行结果如下。

```

12:48:14.319 [main] INFO com.cloudnative.common.RetryCommon - invoke-----
12:48:14.319 [main] INFO com.cloudnative.common.RetryCommon - invokeRetry
retryCount=10

```





```

12:48:15.323 [main] INFO com.cloudnative.common.RetryCommon - invoke-----
12:48:15.323 [main] INFO com.cloudnative.common.RetryCommon - invokeRetry
retryCount=9
12:48:16.428 [main] INFO com.cloudnative.common.RetryCommon - invoke-----
12:48:16.429 [main] INFO com.cloudnative.common.RetryCommon - invokeRetry
retryCount=8
12:48:17.631 [main] INFO com.cloudnative.common.RetryCommon - invoke-----
12:48:17.631 [main] INFO com.cloudnative.common.RetryCommon - invokeRetry
retryCount=7
12:48:18.937 [main] INFO com.cloudnative.common.RetryCommon - invoke-----
12:48:18.937 [main] INFO com.cloudnative.common.RetryCommon - invokeRetry
retryCount=6
12:48:20.342 [main] INFO com.cloudnative.common.RetryCommon - invoke-----
12:48:20.342 [main] INFO com.cloudnative.common.RetryCommon - invokeRetry
retryCount=5
12:48:21.843 [main] INFO com.cloudnative.common.RetryCommon - invoke-----
12:48:21.843 [main] INFO com.cloudnative.common.RetryCommon - invokeRetry
retryCount=4
12:48:23.446 [main] INFO com.cloudnative.common.RetryCommon - invoke-----
12:48:23.446 [main] INFO com.cloudnative.common.RetryCommon - invokeRetry
retryCount=3
12:48:25.150 [main] INFO com.cloudnative.common.RetryCommon - invoke-----
12:48:25.150 [main] INFO com.cloudnative.common.RetryCommon - invokeRetry
retryCount=2
12:48:26.954 [main] INFO com.cloudnative.common.RetryCommon - invoke-----
12:48:26.954 [main] INFO com.cloudnative.common.RetryCommon - invokeRetry
retryCount=1
12:48:28.859 [main] INFO com.cloudnative.common.RetryCommon - invoke-----
12:48:28.859 [main] INFO com.cloudnative.common.RetryCommon - retry completed

```

以上两种重试策略比较容易理解，但是存在一个共同的问题，重试策略和业务处理耦合严重，当然我们可以对这两种策略进行抽象。另外，为了便于理解，并没有加入超时时间这个参数，它可以通过 `Futtrue` 来实现。实际上，重试策略的逻辑是通用的，早已经有人把这个逻辑抽象出来了。下面通过 `Spring-tryer` 和 `Guava-retrying` 工具尝试一下另外两种优雅的重试方案，这两个库都是开源的。

### 3. 基于 Spring-tryer 重试

引入相应依赖的 JAR 包。

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>

```

```

</dependency>
<dependency>
    <groupId>org.springframework.retry</groupId>
    <artifactId>spring-retry</artifactId>
    <version>1.2.2.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.8.6</version>
</dependency>

```

实现 Service。添加@Retryable 和@Recover 注解。

```

@Service
public class SpringRetryService {
    private static org.slf4j.Logger log= LoggerFactory.getLogger(SpringRetryService.
class.getName());
    @Retryable(value= {RemoteAccessException.class},maxAttempts = 3,backoff =
@Backoff(delay = 1000,multiplier = 2))
    public void call() throws Exception {
        log.info("do something.....");
        throw new RemoteAccessException("RPC 调用异常----RemoteAccessException");
    }
    @Retryable(value= {RuntimeException.class},maxAttempts = 3,backoff =
@Backoff(delay = 1000,multiplier = 2))
    public void call2() throws Exception {
        log.info("do something.....");
        throw new RuntimeException("RPC 调用异常----RuntimeException");
    }
    @Retryable(exceptionExpression="#{message.contains('customException')}",
maxAttempts = 3,backoff = @Backoff(delay = 1000,multiplier = 2))
    public void call3() throws Exception {
        log.info("do something.....");
        throw new Exception("RPC 调用异常----customException");
    }
    @Recover
    public void recoverRemoteAccessException(RemoteAccessException e) {
        log.info(e.getMessage());
    }
    @Recover
    public void recoverRuntimeException(RuntimeException e) {
        log.info(e.getMessage());
    }
}

```



```
@Recover
public void recoverCustomException(Exception e) {
    log.info(e.getMessage());
}
}
```

@Retryable 注解修饰方法，被修饰的方法发生异常时会重试。

- value: 指定发生的异常进行重试。上面代码分别模拟了两个异常，call 模拟的是 RemoteAccessException，call2 模拟的是 RuntimeException。
- exceptionExpression: 异常表达式，上面代码 call3 中模拟了包含 customException 关键字的异常执行哪个 recover 方法。
- include: 和 value 一样，默认空，当 exclude 也为空时，所有异常都重试。
- exclude: 指定异常不重试，默认空，当 include 也为空时，所有异常都重试。
- stateful: 标识重试是否有状态的，异常引发事务失效的时候需要注意它，该参数默认为 false。
- maxAttempts: 重试次数，默认为 3，上面我们设置为 2。
- backoff: 退避策略，重试机制，默认没有。

@Backoff 注解，退避策略，重试机制，默认为空。当该参数为空时是，失败立即重试，重试的时候阻塞线程。

- value: 重试延迟时间，默认 1000。
- delay: 等同于 value 参数，两个参数有一个即可。
- maxDelay: 两次重试间的最大间隔时间。当设置 multiplier 参数后，下次延迟时间是根据上次延迟时间乘以 multiplier 得出的，这会导致两次重试间的延迟时间越来越长，该参数限制两次重试的最大间隔时间。当间隔时间大于该值时，计算出的间隔时间将会被忽略，使用上次的重试间隔时间。
- multiplier: 指定延迟的倍数，比如当 delay=1000，multiplier=2 时，第一次重试为 1 秒后，第二次重试为 2 秒，第三次重试为 4 秒。
- random: 是否启用随机退避策略，默认为 false。设置为 true 时启用退避策略，重试延迟时间将是 delay 和 maxDelay 间的一个随机数。设置该参数的目的是重试的时候避免同时发起重试请求，造成 Ddos 攻击。

@Recover 注解，当重试到达指定次数时，被注解的方法将被回调，可以在该方法中进行日志处理或者发送告警。需要注意的是发生的异常和入参类型一致时才会回调。

Main 函数启动实现如下。

```
@Configuration
```



```

@EnableRetry
@EnableAspectJAutoProxy(proxyTargetClass=true)
public class Application {

    public static void main(String[] args) throws Exception {
        ApplicationContext applicationContext = new AnnotationConfigApplication
Context("com.cloudnative.common");
        final SpringRetryService springRetryService = applicationContext.getBean
(SpringRetryService.class);
        springRetryService.call();
        springRetryService.call2();
        springRetryService.call3();
    }
}

```

在 Spring Boot 中启动服务非常方便，扫描指定路径，获取 bean，分别调用相应方法，执行结果如下。

```

16:23:39.621 [main] DEBUG org.springframework.retry.support.RetryTemplate - Retry:
count=0
16:23:39.629 [main] INFO com.cloudnative.common.SpringRetryService - do
something.....
16:23:39.630 [main] DEBUG org.springframework.retry.backoff.ExponentialBackOffPolicy
- Sleeping for 1000
16:23:40.634 [main] DEBUG org.springframework.retry.support.RetryTemplate - Checking
for rethrow: count=1
16:23:40.634 [main] DEBUG org.springframework.retry.support.RetryTemplate - Retry:
count=1
16:23:40.634 [main] INFO com.cloudnative.common.SpringRetryService - do
something.....
16:23:40.634 [main] DEBUG org.springframework.retry.backoff.ExponentialBackOffPolicy
- Sleeping for 2000
16:23:42.635 [main] DEBUG org.springframework.retry.support.RetryTemplate - Checking
for rethrow: count=2
16:23:42.635 [main] DEBUG org.springframework.retry.support.RetryTemplate - Retry:
count=2
16:23:42.635 [main] INFO com.cloudnative.common.SpringRetryService - do
something.....
16:23:42.635 [main] DEBUG org.springframework.retry.support.RetryTemplate - Checking
for rethrow: count=3
16:23:42.635 [main] DEBUG org.springframework.retry.support.RetryTemplate - Retry
failed last attempt: count=3
16:23:42.635 [main] INFO com.cloudnative.common.SpringRetryService - RPC 调用异常
----RemoteAccessException

```



```
16:23:42.636 [main] DEBUG org.springframework.retry.support.RetryTemplate - Retry:
count=0
16:23:42.636 [main] INFO com.cloudnative.common.SpringRetryService - do
something.....
16:23:42.637 [main] DEBUG org.springframework.retry.backoff.ExponentialBackOffPolicy
- Sleeping for 1000
16:23:43.639 [main] DEBUG org.springframework.retry.support.RetryTemplate - Checking
for rethrow: count=1
16:23:43.639 [main] DEBUG org.springframework.retry.support.RetryTemplate - Retry:
count=1
16:23:43.639 [main] INFO com.cloudnative.common.SpringRetryService - do
something.....
16:23:43.639 [main] DEBUG org.springframework.retry.backoff.ExponentialBackOffPolicy
- Sleeping for 2000
16:23:45.643 [main] DEBUG org.springframework.retry.support.RetryTemplate - Checking
for rethrow: count=2
16:23:45.643 [main] DEBUG org.springframework.retry.support.RetryTemplate - Retry:
count=2
16:23:45.644 [main] INFO com.cloudnative.common.SpringRetryService - do
something.....
16:23:45.644 [main] DEBUG org.springframework.retry.support.RetryTemplate - Checking
for rethrow: count=3
16:23:45.644 [main] DEBUG org.springframework.retry.support.RetryTemplate - Retry
failed last attempt: count=3
16:23:45.644 [main] INFO com.cloudnative.common.SpringRetryService - RPC 调用异常
----RuntimeException
16:23:45.654 [main] DEBUG org.springframework.retry.support.RetryTemplate - Retry:
count=0
16:23:45.654 [main] INFO com.cloudnative.common.SpringRetryService - do
something.....
16:23:45.672 [main] DEBUG org.springframework.retry.backoff.ExponentialBackOffPolicy
- Sleeping for 1000
16:23:46.675 [main] DEBUG org.springframework.retry.support.RetryTemplate - Checking
for rethrow: count=1
16:23:46.676 [main] DEBUG org.springframework.retry.support.RetryTemplate - Retry:
count=1
16:23:46.677 [main] INFO com.cloudnative.common.SpringRetryService - do
something.....
16:23:46.677 [main] DEBUG org.springframework.retry.backoff.ExponentialBackOffPolicy
- Sleeping for 2000
16:23:48.679 [main] DEBUG org.springframework.retry.support.RetryTemplate - Checking
for rethrow: count=2
16:23:48.679 [main] DEBUG org.springframework.retry.support.RetryTemplate - Retry:
```

```

count=2
16:23:48.679 [main] INFO com.cloudnative.common.SpringRetryService - do
something.....
16:23:48.679 [main] DEBUG org.springframework.retry.support.RetryTemplate - Checking
for rethrow: count=3
16:23:48.679 [main] DEBUG org.springframework.retry.support.RetryTemplate - Retry
failed last attempt: count=3
16:23:48.680 [main] INFO com.cloudnative.common.SpringRetryService - RPC 调用异常
----customException

```

#### 4. 基于 Guava-retrying 重试

Guava Retrying 是一个灵活方便的重试组件，包含了多种重试策略，而且扩展起来非常容易。它是 Google Guava 库的一个扩展，允许为任意函数调用创建可配置的重试策略，并且监控每次重试的结果和行为。它还可以显著提升远程服务调用的可用性。

下面通过代码来说明如何采用 Guava Retrying 实现重试。

首先，在 pom 中引入 guava-retrying 包。

```

<dependency>
  <groupId>com.github.rholder</groupId>
  <artifactId>guava-retrying</artifactId>
  <version>2.0.0</version>
</dependency>

```

然后，定义实现。我们模拟一个场景，远程调用 API 接口 `http://localhost:8080/products/1`，出现异常进行重试。通过 Builder 实例化一个 Retrier 类，Retrier 类可以定义重试的时机、策略等。

```

public void call() {
    Retrier<Boolean> retryer = RetrierBuilder.<Boolean>newBuilder()
        .retryIfException() //抛出 runtime 异常、checked 异常时都会重试，但是抛出 error
不会重试
        .retryIfResult(Predicates.equalTo(false)) //返回 false 需要重试
        .withWaitStrategy(WaitStrategies.incrementingWait(2, TimeUnit.SECONDS, 4, T
imeUnit.SECONDS)) //重试策略，初试 2 秒，以后增加 4 秒
        .withStopStrategy(StopStrategies.stopAfterAttempt(3)) //重试次数
        .build();

    try {
        //重试入口采用 call 方法。因为它用的是 java.util.concurrent.Callable<V>的 call 方法，
所以执行是线程安全的
        boolean result = retryer.call(new Callable<Boolean>() {
            @Override

```



```
public Boolean call() throws Exception {
    try {
        log.info("----call----");
        String url = "http://localhost:8080/products/1";
        RestTemplate restTemplate=new RestTemplate();
        Product product = restTemplate.getForEntity(url,
Product.class).getBody();
        log.info("-----"+product.toString());
        if (product!=null){
            return true;
        }
        //特别注意: 返回 false 说明需要继续重试, 返回 true 则说明不用重试
        return false;
    } catch (Exception e) {
        log.error(e.getMessage(),e);
        throw new Exception(e);
    }
}

});

} catch (ExecutionException e) {
    log.error(e.getMessage());
} catch (RetryException ex) {
    log.error(ex.getMessage());
}
}
```

需要重点说明的相关内容如下。

### (1) RetryerBuilder 创建者

RetryerBuilder 是一个 factory 创建者, 可以创建重试者 Retryer 实例, 定义各种策略。RetryerBuilder 的重试源支持 Exception 异常对象和自定义断言对象, 通过 retryIfException 和 retryIfResult 设置, 同时支持多个且能兼容。

### (2) RetryIf 重试源定义

- retryIfException: 抛出 runtime 异常、checked 异常时都会重试, 但是抛出 error 时不会重试。当然也可以通过 Predicate 实现。
- retryIfRuntimeException: 只会在抛出 runtime 异常时重试, 抛出 checked 异常和 error 时都不重试。
- retryIfExceptionOfType: 允许只在发生特定异常的时候才重试, 比如 NullPointerException 和 IllegalStateException 都属于 runtime 异常, 也包括自定义的



error。

- `retryIfResult`：指定 `Callable` 方法在返回值的时候进行重试，如 `retryIfResult(Predicates.equalTo(false))`，返回 `false` 时重试。

### (3) WaitStrategy 等待策略

- `WaitStrategies.noWait()`：失败后立刻重试，没有等待时间。
- `WaitStrategies.fixedWait()`：固定等待时间之后重试，如每隔 3 秒重试一次可以定义为 `WaitStrategies.fixedWait(3, TimeUnit.SECONDS)`。
- `WaitStrategies.randomWait()`：随机等待时间之后重试，如等待 0~10 秒随机时间之后进行重试，可以定义为 `WaitStrategies.randomWait(10, TimeUnit.SECONDS)`。  
`WaitStrategies.randomWait(10, TimeUnit.SECONDS, 50, TimeUnit.SECONDS)`表示等待最小值和最大值之间的随机值之后重试。
- `WaitStrategies.incrementingWait()`：递增等待时长策略（提供一个初始值和步长，等待时间随重试次数增加而增加）。
- `WaitStrategies.fibonacciWait()`：Fibonacci 等待时长策略，按照斐波那契数列时长等待。
- `WaitStrategies.exponentialWait()`：指数等待时长策略，按照指数递增（2<sup>n</sup>）来等待。它是一个简单的等待策略。
- `WaitStrategies.exceptionWait()`：异常时长等待策略，根据抛出的异常类型来决定等待多长时间。

### (4) StopStrategy 停止策略

- `StopStrategies.stopAfterDelay()`：设定一个最长允许的执行时间，如 `StopStrategies.stopAfterDelay(10, TimeUnit.SECONDS)`表示设定最长执行 10 秒，不考虑任务执行次数，只要重试的时长超出了最长时间，就终止任务并返回重试异常 `RetryException`。
- `StopStrategies.neverStop()`：永远不停止，用于需要一直轮询直到返回期望结果的情况。
- `StopStrategies.stopAfterAttempt()`：设定最大重试次数，如果超出最大重试次数则停止重试，并返回重试异常。例如 `StopStrategies.stopAfterAttempt(3)`表示最多重试 3 次。

### (5) 任务阻塞策略

**BlockStrategies**：在阻塞的时间具体做什么，阻塞时间是指从当前任务执行完到下次任务开始前这段时间。例如第一次调用失败，当进行重试之前，需要等待 1 秒再进行下次调用，那么这 1 秒的等待有多种实现，如 `sleep`、`锁`、`wait` 等，默认策略为 `BlockStrategies.THREAD_SLEEP_STRATEGY`，也就是调用 `Thread.sleep()`。





### (6) AttemptTimeLimiters 单次限时

`AttemptTimeLimiters.fixedTimeLimit()`: 单次任务执行时长限制。如果超时, 则终止执行当前任务。如 `AttemptTimeLimiters.fixedTimeLimit(1, TimeUnit.SECONDS)` 表示单次任务最长执行 1 秒, 如果超出, 则可以抛出 `TimeoutException` 异常。

### (7) Listener 监听器

**RetryListener**: 自定义重试监听器, 当发生重试之后, 可以用于异步记录错误日志或者发送错误告警等。每次重试之后, `guava-retrying` 会自动回调我们注册的监听。我们可以注册多个 `RetryListener`, 它们会按照注册顺序依次调用。

- 正常时的运行结果如下所示。

```
16:08:29.911 [main] INFO com.cloudnative.common.GuavaRetryService - ----call-----
16:08:30.163 [main] DEBUG org.springframework.web.client.RestTemplate - Created GET
request for "http://localhost:8080/products/1"
16:08:30.198 [main] DEBUG org.springframework.web.client.RestTemplate - Setting
request Accept header to [application/json, application/*+json]
16:08:30.211 [main] DEBUG org.springframework.web.client.RestTemplate - GET request
for "http://localhost:8080/products/1" resulted in 200 (OK)
16:08:30.212 [main] DEBUG org.springframework.web.client.RestTemplate - Reading
[class com.cloudnative.dto.Product] as "application/json;charset=UTF-8" using
[org.springframework.http.converter.json.MappingJackson2HttpMessageConverter@2fd66ad3
]
16:08:30.226 [main] INFO com.cloudnative.common.GuavaRetryService -
-----Product{id=1, name='watch1', count=11, desc='watch desc1'}
```

- 异常时的运行结果如下所示。

```
16:13:13.533 [main] INFO com.cloudnative.common.GuavaRetryService - ----call-----
16:13:13.785 [main] DEBUG org.springframework.web.client.RestTemplate - Created GET
request for "http://localhost:8080/producwts/1"
16:13:13.819 [main] DEBUG org.springframework.web.client.RestTemplate - Setting
request Accept header to [application/json, application/*+json]
16:13:13.831 [main] DEBUG org.springframework.web.client.RestTemplate - GET request
for "http://localhost:8080/producwts/1" resulted in 404 (Not Found); invoking error
handler
16:13:13.834 [main] ERROR com.cloudnative.common.GuavaRetryService - 404 Not Found
16:13:15.839 [main] INFO com.cloudnative.common.GuavaRetryService - ----call-----
16:13:15.847 [main] DEBUG org.springframework.web.client.RestTemplate - Created GET
request for "http://localhost:8080/producwts/1"
16:13:15.848 [main] DEBUG org.springframework.web.client.RestTemplate - Setting
request Accept header to [application/json, application/*+json]
16:13:15.852 [main] DEBUG org.springframework.web.client.RestTemplate - GET request
```





```
for "http://localhost:8080/producwts/1" resulted in 404 (Not Found); invoking error handler
16:13:15.854 [main] ERROR com.cloudnative.common.GuavaRetryService - 404 Not Found
16:13:21.855 [main] INFO com.cloudnative.common.GuavaRetryService - ----call-----
16:13:21.861 [main] DEBUG org.springframework.web.client.RestTemplate - Created GET request for "http://localhost:8080/producwts/1"
16:13:21.862 [main] DEBUG org.springframework.web.client.RestTemplate - Setting request Accept header to [application/json, application/*+json]
16:13:21.866 [main] DEBUG org.springframework.web.client.RestTemplate - GET request for "http://localhost:8080/producwts/1" resulted in 404 (Not Found); invoking error handler
16:13:21.867 [main] ERROR com.cloudnative.common.GuavaRetryService - 404 Not Found
16:13:21.867 [main] ERROR com.cloudnative.common.GuavaRetryService - Retrying failed to complete successfully after 3 attempts.
```

### 4.3.6 隔离策略

隔离是为了在系统发生故障时，限制传播范围和影响范围，特别要注意非核心系统的故障对核心系统的影响。如电商举办秒杀活动时，系统瞬时压力非常大，在极端的时间内对库存进行扣减、查询，会导致正常业务受到影响，因此一般都会把秒杀隔离出来。

**线程池隔离。**在单体架构中，为了保护核心业务流程不受干扰，通常会使用线程池隔离。对于核心的业务流程，需要重点保护，它们可以独占一个线程池。当然，在微服务架构中，这种隔离方式仍然可以继续使用。

**进程隔离。**在微服务架构中，可以把核心流程独立成一个服务。通过进程隔离，每个服务可以部署到不同的容器或虚拟机内，形成更高级别的隔离。

**集群隔离。**随着系统规模的逐步扩大，可能会发生这种情况，即给终端用户使用的服务和管理员使用的服务需要隔离。因为有可能后台一个操作会影响终端用户，这种情况可以通过接口限流的方式隔离，或者是部署另外一个集群。

**用户隔离。**在很多系统中，某用户的特殊操作可能会影响到其他用户，这时候我们就要从用户的角度进行隔离。如某电商运营人员批量修改了价格数据，数据进入分布式消息中间件，其他商家修改价格的操作就会被阻塞，需要排队。如果能够根据条数或者用户进行隔离，那么用户体验会更好。

**租户隔离。**租户是用户的特殊形式，对隔离性要求更高。通常租户隔离主要有以下三种方式。

- 逻辑隔离。在数据库加字段隔离的不彻底，受影响的概率高，但是实现简单，节省资源。
- 物理隔离。部署独立的服务和数据库，划分独立的带宽，隔离比较彻底，但是需要



更多的资源成本。

- 混合隔离。逻辑隔离和物理隔离的混合，需要根据业务场景决定什么地方需要逻辑隔离，什么地方需要物理隔离。

通过 **Hystrix** 实现隔离。Hystrix 组件提供了两种隔离解决方案：线程池隔离和信号量隔离。两种隔离方案都是限制对共享资源的并发访问量，线程在就绪状态、运行状态、阻塞状态、终止状态间转变时需要由操作系统调度，占用很大的性能消耗，而信号量是在访问共享资源时进行 tryAcquire，tryAcquire 成功才允许访问共享资源。我们通过以下代码来进行说明。

```
public HelloCommandIsolateThreadPool(String name) {
    super(HystrixCommand.Setter. //设置 GroupKey 用于 dashboard 分组展示
        withGroupKey(HystrixCommandGroupKey.Factory.asKey("hello"))
        //设置 commandKey 用户隔离线程池，不同的 commandKey 会使用不同的线程池
        .andCommandKey(HystrixCommandKey.Factory.asKey("hello" + name))
        //设置线程池名字的前缀，默认使用 commandKey
        .andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("hello$Pool" + name))
        //设置线程池相关参数
        .andThreadPoolPropertiesDefaults(HystrixThreadPoolProperties.Setter()
            .withCoreSize(15)
            .withMaxQueueSize(10)
            .withQueueSizeRejectionThreshold(2))
        //设置 command 相关参数
        .andCommandPropertiesDefaults(HystrixCommandProperties.Setter()
            //是否开启熔断器机制
            .withCircuitBreakerEnabled(true) //舱壁隔离策略
            .withExecutionIsolationStrategy(HystrixCommandProperties.ExecutionIsolationStrategy.THREAD) //circuitBreaker 打开后多久关闭
            .withCircuitBreakerSleepWindowInMilliseconds(5000)));
}
```

### 4.3.7 熔断器

由于微服务架构中存在大量的远程通信，而网络是脆弱的，基础服务的故障可能会导致级联故障，进而造成了整个系统的不可用，这种现象被称为服务雪崩效应。服务雪崩效应描述的是一种因服务提供者的不可用导致服务消费者不可用，并将不可用逐渐放大的过程。

例如，当某个用户在网提交一个订单的时候，在微服务架构中，可能这个请求要经过  $N$  个服务。假设某个库存服务实例的 CPU 非常繁忙，导致请求一直阻塞，我们可能想到的一个解决办法是设置超时时间，那这个超时时间设置多少合适呢？如果设置的超时时间



过短，消费者（被调用的服务）没有处理完，提供者又发过来一次请求，那么最终导致消费者崩溃。如果设置的过长，那么用户发现几秒都没有响应，可能会以为网络太慢，又重新提交一次订单。那么有没有更好的解决办法呢？下面我们就介绍一种模式来解决这个问题。

熔断器模式（Circuit Breaker Pattern）的原理类似于家里的电路熔断器的原理。当发生短路或者超负荷时，熔断器能够主动熔断电路，以避免灾难发生。在分布式系统中应用熔断器模式后，当消费者发生大量超时，提供者能够主动熔断，以防止服务被进一步拖垮。经过一段时间，当情况开始变好后，消费者重新进行尝试，提供者又能重新提供服务，这就是所谓的 Design for failure，系统有自恢复能力。

图 4-10 是对于熔断器模式各个状态转换的说明。

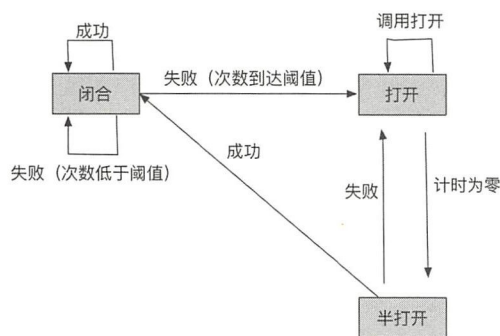


图 4-10 熔断器模式各个状态转换说明

熔断器的三个状态如下。

- 打开（open）：熔断器打开状态。消费者的请求禁止通过熔断器调用生产者，调用快速失败。
- 闭合（closed）：熔断器闭合状态。消费者的请求可以顺利通过，到达生产者。熔断器会记录最近调用失败次数，当达到阈值时，状态切换到熔断器打开状态。然后，熔断器开始倒计时，当剩余时间为 0 时，则切换到半打开（half open）状态。这个倒计时实际上是给生产者一个自动恢复的时间。
- 半打开：熔断器半打开状态。熔断器允许消费者的部分请求通过熔断器尝试调用生产者，因为前面的失败有可能是因为断网等原因导致的。如果这些请求顺利调用成功，则熔断器恢复到闭合状态；如果这些请求仍然调用失败，则切换到打开状态，重新开始计时。

Spring Cloud Hystrix 提供了熔断器、线程隔离等一系列服务保护的能力，使用起来非





常简单,引入依赖的JAR包,通过简单的注解即可实现。下面我们在2.10节基于Spring Cloud实现的Demo上进行修改,其中eureka-registry和eureka-provider保持不变。

首先,引入JAR包。

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
```

然后,在eureka-consumer上进行修改,启动类需要增加@EnableHystrix注解打开Hystrix。

```
@EnableDiscoveryClient
@SpringBootApplication
@EnableHystrix
public class ConsumerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConsumerApplication.class, args);
    }
    @Bean
    @LoadBalanced
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

最后,Controller修改如下。

```
@RestController
public class ProductDetailController {
    @Autowired
    RestTemplate restTemplate;

    @RequestMapping(value = "/detail/{id}", method = RequestMethod.GET)
    @HystrixCommand(fallbackMethod = "getFallback",
        commandProperties = {
            @HystrixProperty(name = "execution.isolation.thread.
timeoutInMilliseconds", value = "500"),
            @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold",
value = "2"),
        }
    )
    public String getProductDetail(@PathVariable long id) {
        System.out.println("getProductDetail:id = [" + id + "]);
```



```
        return restTemplate.getForEntity("http://PROVIDER/price/"+id,String.class).
getBody();
    }

    protected String getFallback(long id,Throwable throwable) {
        System.out.println("getFallback:id = [" + id + "], throwable = [" + throwable
+ " ]");
        return "error" +id;
    }
}
```

fallbackMethod = "getFallback"表示熔断时会调用这个方法，我们可以在类中直接定义出来，两者的返回值和参数要相同。fallback 方法可以额外传入 Throwable 中，方法内根据异常信息进行相应处理。

通过 commandProperties 对熔断器进行配置，如示例中 name ="execution.isolation.thread.timeoutInMilliseconds", value = "500"用来指定超时时间，超过 500 毫秒就会调用 getFallback 方法。name = "circuitBreaker.requestVolumeThreshold", value = "2"表示一个统计窗口内熔断触发的最小个数，默认为 10，在一个统计窗口内达到此阈值就会熔断。还可以通过 circuitBreaker.sleepWindowInMilliseconds 设置熔断多少秒后去尝试请求，通过 circuitBreaker.errorThresholdPercentage 设置失败率达到多少百分比后熔断。还有很多参数，此处不再一一列举。

假设通过修改提供者 eureka-provider，在调用方法中通过 Thread.sleep(600)模拟超时，那么运行结果如下，当第三次进行访问的时候，发现已经出现了短路，熔断器处于打开状态。

```
getProductDetail:id = [1]
getFallback:id      =      [1],      throwable      =      [com.netflix.hystrix.exception.
HystrixTimeoutException]
getProductDetail:id = [1]
getFallback:id      =      [1],      throwable      =      [com.netflix.hystrix.exception.
HystrixTimeoutException]
getFallback:id = [1], throwable = [java.lang.RuntimeException: Hystrix circuit
short-circuited and is OPEN]
```

当使用熔断器时，需要重点关注如下几个问题。

- 熔断器打开时，生产者如何应对？此时应该视业务场景而定，尝试调用其他实例服务；快速失败；业务降级，不显示或者显示缓存值。
- 和其他运维监控系统关联使用。应该在界面上预留开关，可以手动控制熔断器状态。例如，当生产者发生故障的时候，可以通过监控系统发现系统是否已经恢复，如果已经恢复，则可以直接通过预留接口恢复调用，而不必等待重试到指定阈值。当生



产者因为满负荷导致无法响应的时候，应该停止重试。升级时，也可以通过开关进行手动控制。

- 禁止通过一个熔断器控制多个服务。用一个熔断器控制多个服务，可能会导致一个服务出现问题，熔断所有服务的情况发生。

## 4.4 流控设计

### 4.4.1 限流算法

限流很好理解，也就是调节数据流的平均速率，通过限制速率保护自己。例如，当我们不进行限流的时候，系统很可能被压垮；当我们限流时，虽然超出部分被拒绝，但是系统能力范围内的请求可以被正常处理。

另外，限流对于对外开放的接口尤其重要。当系统存在多个租户的时候，需要针对每个租户进行限制，以保障其他租户的体验不受影响。

有很多算法可以实现限流，以下给大家介绍几种常用算法。

#### 固定窗口算法（fixed window）

对于限流来说，最简单的方法就是通过一个变量记录单位时间内的访问次数。固定窗口算法，如图 4-11 所示。通过一个变量 `counter` 记录请求次数，如果一分钟内，请求次数超过 1000，则拒绝 1000 以后发过来的所有请求，1 分钟后 `counter` 会自动归零。如果推敲，则不难发现，这种算法存在漏洞，在 `counter` 临近归零的前后，相当于可以并发请求 2000 次。如果系统只能承受 1000，那么相当于没有起到保护作用。

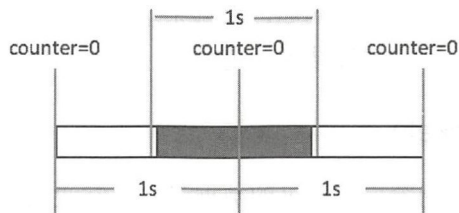


图 4-11 固定窗口算法

实际上，在真实场景中，这种算法效果并不理想。很多消费者调用失败等待 `counter` 刷新，一旦 `counter` 刷新成功，大量的消费请求涌入，会形成踩踏事件。



## 漏桶算法（Leaky Bucket）

漏桶算法主要目的是控制数据注入网络的速率，平滑网络上的突发流量。漏桶算法提供了一种机制，通过它，突发流量可以被重新分配以便为网络提供一个稳定的流量。漏桶算法如图 4-12 所示。



图 4-12 漏桶算法

漏桶算法简单描述如下。

- 水（请求）先进入漏桶（队列）里。
- 漏桶（队列）以一定的速度出水（请求）。
- 水（请求）过大会直接溢出（丢弃数据包）。

漏桶算法能强行限制数据的传输速率，输入速率可以变化，但是输出速率保持不变。

漏桶算法可以看成是一个先进先出的队列（FIFO）。当队列填满的时候，抛弃新请求，队列不保证某个时间点内请求一定会得到处理。往往当消费者一端处理能力有限的时候，通过消息队列起到削峰填谷的作用。例如，在秒杀场景中，如果最后只秒杀一个商品，那么只要队列保留适当请求，就能保证结果一定成功，而不必处理所有请求。

## 令牌桶算法（token bucket）

令牌桶控制的是一个时间窗口内通过的数据量，通常我们会以 QPS、TPS 来衡量，如图 4-13 所示。

令牌桶算法简单描述如下。

- 每秒会有  $x$  个令牌放入桶中，或者说，每过  $1/x$  秒桶中增加一个令牌。
- 桶中最多存放  $n$  个令牌，如果桶满了，则新放入的令牌会被丢弃。
- 当一个  $m$  字节的数据包到达时，消耗  $m$  个令牌，然后发送该数据包。

- 如果桶中可用令牌小于  $m$  个，则该数据包将被缓存或丢弃。

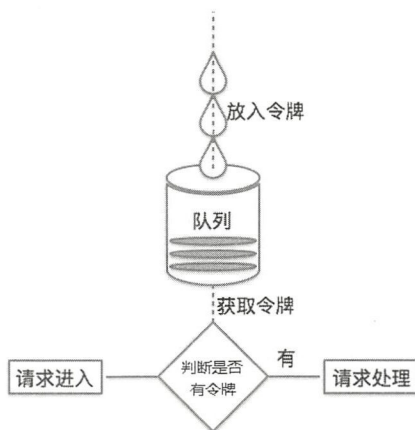


图 4-13 令牌桶算法

漏桶算法和令牌桶算法的对比，如表 4-3 所示。漏桶算法和令牌桶算法的主要区别在于漏桶算法能够强行限制数据的传输速率，而令牌桶算法除能够限制数据的平均传输速率以外，还允许某种程度的突发传输。在令牌桶算法中，只要令牌桶中存在令牌，就允许突发传输数据，直到达到用户配置的上限为止，所以它适合于具有突发特性的流量。

表 4-3 漏桶算法和令牌桶算法的对比

漏桶算法	令牌桶算法
不依赖令牌，不保存令牌	依赖令牌
如果桶满了，则丢弃数据包	如果桶满了，则丢弃令牌，不丢弃数据包
不允许突发，恒定速率	允许突发

#### 4.4.2 流控策略

在分布式系统中，每个环节都要考虑流控。特别是在微服务架构下，服务的数量很多，每个服务根据业务场景的不同，所能承受的请求压力的差距也比较大，所以限流不能一概而论，需要根据压测结果、生产环境上的表现，对每个服务进行设定。一般采用经验值，原则是宁愿设置得过小，也绝对不设置太大。

通常，在下面几个比较重要的节点需要着重考虑。

**请求入口处。**因为此处限流效果最好，当出现大量的无效请求时，后端服务不会受到影响，如通过 Nginx 限流。

**业务服务入口处。**可以依赖于微服务框架，对每个服务节点进行限流，限流并不意味着只能设置单位时间请求数，通过设置最大连接数、最大线程数等都可以实现限流。每个业务服务都应该声明 SLA，包括单位时间处理请求的能力。

**公共基础服务处。**因为公共基础服务太重要，一旦宕机后果严重，所以更需要保护，如数据库、缓存等。

如果我们部署了多个服务实例，只针对一个实例进行限制，效果并不理想，可能会导致有的节点“撑死了”，有的节点“饿死了”。对外则表现为一会儿可以访问，一会儿又访问不了了。解决问题的方法是，在负载均衡配置转发策略，同一 IP 转发到同一服务节点，这样至少对于一个用户来说是一致的，但是对于不同用户来说还是存在问题。如果有一个分布式的限流服务针对同一用户记录访问次数，那么效果会比较好，通常会使用 Redis 来实现。

### 4.4.3 基于 Guava 限流

Guava 是 Google 提供的 Java 扩展类库，其中的限流工具类 `RateLimiter` 采用的就是令牌桶算法，使用起来非常简单。`RateLimiter` 经常用于限制对一些物理资源或者逻辑资源的访问速率，它支持两种获取 `permits` 接口的方法，一种是如果拿不到则立刻返回 `false`，一种会阻塞等待一段时间看能不能拿到。`RateLimiter` 和 Java 中的信号量（`java.util.concurrent.Semaphore`）类似，`Semaphore` 通常用于限制并发量。

我们先看看如何创建一个 `RateLimiter` 实例。

利用 `RateLimiter.create(double permitsPerSecond)` 创建一个每秒包含 `permitsPerSecond` 个令牌的令牌桶，可以理解为 QPS 最多为 `permitsPerSecond`，示例代码如下。

```
//创建桶容量为 5 个且每秒新增 5 个的令牌桶，即每隔 200 毫秒新增一个令牌
RateLimiter limiter = RateLimiter.create(5);
for (int i = 0; i < 5; i++) {
    //消费令牌，如果当前桶中有足够令牌则成功（返回 0）。如果桶中没有令牌则阻塞，最终返回等待时间，假设发令牌间隔是 200 毫秒，则等待 200 毫秒后再去消费令牌
    System.out.println("i = [" + i + "] time = [" + limiter.acquire() + "]");
}
```

运行结果如下。

```
i = [0] time = [0.0]
i = [1] time = [0.198213]
i = [2] time = [0.198577]
i = [3] time = [0.196172]
i = [4] time = [0.198908]
```



第 1 个立刻拿到，后面 4 个差不多每隔 200 毫秒拿到 1 个，此时实现了平均速率。但是我们知道令牌桶是支持突发的，且看下面这段代码。

```
//创建桶容量为 5 个且每秒新增 5 个的令牌桶，即每隔 200 毫秒新增一个令牌
RateLimiter limiter = RateLimiter.create(5);
//第一次把 5 个令牌全部拿完
System.out.println("first time = ["+limiter.acquire(5)+"]");
System.out.println("second time = ["+limiter.acquire(1)+"]");
System.out.println("third time = ["+limiter.acquire(5)+"]");
```

运行结果如下。

```
first time = [0.0]
first time = [0.998358]
first time = [0.195814]
```

结果证明，令牌桶是允许突发的，第一次就全部拿完所有的令牌，第二次需要等待 1 秒。但是，这不禁让大家产生了疑惑，如果允许这种突发，就会和前面讲到的固定窗口算法一样，突发流量会带来踩踏事件。显然，Guava 不会忘记这个问题。实际上，RateLimiter 提供了两种令牌桶算法实现：平滑突发限流（SmoothBursty）和平滑预热限流（SmoothWarmingUp）。上面介绍的是平滑突发限流，下面我们来看一下平滑预热限流。

平滑预热限流实现起来很简单，只需要在 create 方法中传递更多的参数。通过 RateLimiter create(double permitsPerSecond, long warmupPeriod, TimeUnit unit) 创建一个每秒包含 permitsPerSecond 个令牌的令牌桶，可以理解为 QPS 最多为 permitsPerSecond，并包含某个时间段的预热期，示例代码如下。

```
RateLimiter limiter = RateLimiter.create(5, 1, TimeUnit.SECONDS);
for(int i = 0; i < 5; i++) {
    System.out.println("i = [" + i + "] time = ["+limiter.acquire()+"]");
}
```

运行结果如下。

```
i = [0] time = [0.0]
i = [1] time = [0.518329]
i = [2] time = [0.354786]
i = [3] time = [0.217197]
i = [4] time = [0.197893]
```

结果证明，令牌桶进行了缓冲，逐步达到平均速率。

另外，在某些场景中，我们希望如果能获取到令牌就获取，如果获取不到，则丢弃请求，不必一直阻塞等待。因此，RateLimiter 还提供了 tryAcquire 能力，尝试获取令牌。如

果能取到则返回 `true`，取不到则返回 `false`，详细代码如下所示。

```
boolean tryAcquire(); // 尝试获取一个令牌
boolean tryAcquire(int permits); // 尝试获取 permits 个令牌
boolean tryAcquire(long timeout, TimeUnit unit); // 尝试获取一个令牌，最多等待 timeout
时间
boolean tryAcquire(int permits, long timeout, TimeUnit unit); // 尝试获取 permits 个
令牌，最多等待 timeout 时间
```

#### 4.4.4 基于 Nginx 限流

因为 Nginx 通常作为边缘服务的入口，所以在 Nginx 上限流会取得比较好的效果，下面介绍一下基于 Nginx 限流的方案。

##### 连接数限流模块 ngx\_http\_limit\_conn\_module

ngx\_http\_limit\_conn\_module 的作用是限制并发访问，包括异常流量、突发流量，甚至是恶意攻击等，它可以根据定义的键来限制每个键值的连接数。

ngx\_http\_limit\_conn\_module 提供了 `limit_conn_zone` 和 `limit_conn` 两个配置参数。其中，`limit_conn_zone` 只能配置在 `http{}` 段，而 `limit_conn` 则可以配置于 `http{}`、`server{}`、`location{}` 中。

我们可以在 `nginx_conf` 的 `http{}` 中加上如下配置实现限制每个用户的并发连接数。其中，`$binary_remote_addr` 表示使用真实 IP 地址作为键；`req_one` 是我们定义的名字，需要与 `server{}` 对应；`rate=100r/s` 表示每秒允许 100 个请求。

```
limit_conn_zone $binary_remote_addr zone=req_one:10m rate=100r/s;
```

配置记录被限流后的日志级别，默认为 `error` 级别。

```
limit_conn_log_level error;
```

配置被限流后返回的状态码，默认返回 503。

```
limit_conn_status 503;
```

然后在 `server{}` 里加上如下代码。

```
limit_conn req_one 10;
```

上述代码限制每个 IP 并发连接数为 10，这里也可以配置多个，例如读的并发连接数是 10，写的并发连接数是 1。

## 请求限制模块 ngx\_http\_limit\_req\_module

ngx\_http\_limit\_conn\_module 的作用是限制并发访问，而 ngx\_http\_limit\_req\_module 可以为我们限制请求数，该模块可以通过定义的键值来限制请求处理的频率。例如，可以限制来自单个 IP 地址的请求处理频率。该方法使用了漏桶算法，限制每秒固定处理请求数。如果请求的频率超过了限制域配置的值，那么请求处理会被延迟或被丢弃，因此所有的请求都是以固定配置的频率被处理的。

首先，在 nginx\_conf 的 http{} 中加上如下配置实现限制。

```
limit_req_zone $binary_remote_addr zone= req_one:10m rate=100r/s;
```

其中，\$binary\_remote\_addr 表示使用真实 IP 地址作为键；req\_one 是我们定义的名字，需要与 server{} 对应；rate=100r/s 表示每秒允许 100 个请求。

然后，在 server{} 里加上如下代码。

```
limit_req zone= req_one burst=5;
```

设置桶的容量为 5，默认为 0。如果配置 nodelay，则允许突发，否则保持固定速率。

## 4.5 容量预估

孙子曰：“知己知彼，百战不殆。”

我们需要知道当前系统能够承受多大压力。传统预测方式是在测试环境下进行的，针对场景进行数据模拟，需要开发、测试人员根据线上的场景，评估可能出现的情况。根据压测结果分析瓶颈点，这种方式更依赖于人的经验。常用的工具包括 Apache Benchmark、LoadRunner、Jmeter 等。但是这种做法非常不准确，很难模拟出接近生产环境的场景和数据。压测的质量非常依赖开发人员、测试人员的水平和认真程度。

互联网公司普遍采用全链路压测的方式，如京东的 ForceBot、阿里巴巴的全链路压测平台。生产环境压测，如图 4-14 所示。全链路压测平台在请求入口进行真实流量复制，开源实现可以参考 TCPCopy。这样可以简化模拟数据带来的成本，将复制的请求引入压测环境，对压测环境的服务进行压力测试。为了加大压力，可以通过 TCPCopy 的参数调节，也可以通过 MQ 等工具“蓄水”。在数据库一侧通过影子表进行隔离，影子表和生产表建立相同的数据结构，通过后缀进行区分，便于隔离删除。如果害怕对生产环境造成影响，那么可以把生产表和影子表放到不同的数据库中。



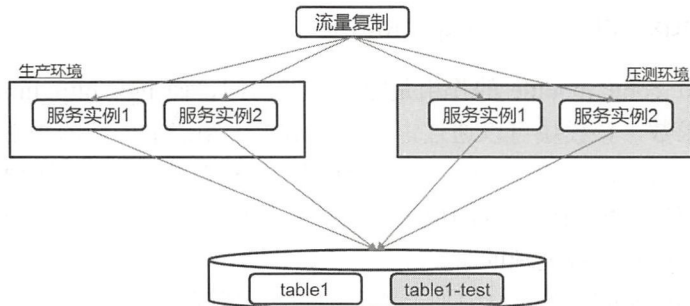


图 4-14 生产环境压测

全链路压测需要注意以下几个点。

- 找到核心流程。做全链路压测需要巨大的成本，不可能全做。一个系统中的核心流程可能只有 20%，这是压测的目标。
- 选择隔离方式。一种是独立的环境进行压测，隔离效果好，简单方便，但是资源成本高。另一种是和生产环境混合，通过参数进行识别，在框架和服务处进行特殊处理，这种方式真实性更高，节省资源，但是隔离性不好。
- 缩小依赖服务范围。

思考：如果对 A 服务进行压测，那么 A 服务依赖的服务应该如何配合。

## 4.6 故障演练

我们害怕发生故障，因此制定了很多应对策略，但是这些策略在没有测试的情况下谁也不敢轻易启用，害怕引起更严重的故障。例如，某互联网公司因为网络原因导致大规模故障，中断几个小时，是因为没有灾备吗？当然不太可能。虽然他们做了灾备，但是因为很长时间没有测试过了，所以并不敢切换，灾备成了一个应付领导的噱头。故障演练正是为了解决“不敢切换”的问题。

在这方面，Netflix 一直走在前列。早在 2010 年的一篇博文中<sup>①</sup>，Netflix 的工程师 John Ciancutti 就有一句经典的话：“The best way to avoid failure is to fail constantly”，大意就是，避免失败最好的方式就是不断失败。2012 年，Netflix 开源了 Chaos Monkey，Chaos Monkey 是一个在生产环境随机选择并关闭服务的工具。对于这个选择，有人会觉得很疯狂，因为这些演练可能会对最终用户产生影响，但是通过这种频繁的演练，能够使开发人员对服务稳定性有更高的重视，更加关注 Design for failure，以确保不会因为故障对最终用户产生更

<sup>①</sup> 博文地址 <https://medium.com/netflix-techblog/5-lessons-weve-learned-using-aws-1f2a28588e4c>。

大的影响。可以在 Chaos Monkey 上配置执行计划，默认只在工作日的上午 9 点至下午 3 点执行。

Chaos Monkey 属于 Netflix 的 Simian Army 产品中的一员，Simian Army 由一组工具构成，包括如下成员。

- Chaos Monkey: 随机关闭生产环境中的实例。
- Latency Monkey: 让某台机器的请求或返回变慢，观察系统的表现，可以用来测试上游服务是否有服务降级能力，当然如果响应时间特别长，也就相当于服务不可用。
- Chaos Gorilla: 模拟 AZ 故障，中断一个机房，验证是否跨可用区部署，业务容灾和恢复的能力。
- Conformity Monkey: 查找不符合最佳实践的实例，并将其关闭。例如，如果某个实例不在自动伸缩组里，那么就将其关闭，观察服务是否能够重新正常启动。
- Doctor Monkey: 查找不健康实例的工具，除了运行在每个实例上的健康检查，还会监控外部健康信号，一旦发现不健康实例就会将其移出服务组。
- Janitor Monkey: 查找不再需要的资源，并将其回收，这能在一定程度上降低云资源的浪费。
- Security Monkey: 是 Conformity Monkey 的一个扩展，检查系统的安全漏洞，同时也会保证 SSL 和 DRM 证书仍然有效。
- 10-18 Monkey: 进行本地化及国际化的配置检查，确保不同地区、使用不同语言和字符集的用户都能正常使用 Netflix。

随后，阿里巴巴也开始进行故障演练，它的工具名称叫 MonkeyKing，为每年的“双 11”活动做准备。MonkeyKing 可以模拟硬件故障、API 故障、分布式故障、数据库故障等。

故障演练不仅可以检验业务应用处理失败的能力，也可以用于当故障发生时，如何快速有效地发现并定位故障，通知相应开发人员测试监控的范围及处理故障的时长。

## 4.7 数据迁移

Code is easy, state is hard.

——Edson Yanaga

因为大多数服务是从单体架构开始，伴随着业务的发展逐步拆分，所以会涉及数据的迁移。我们比较提倡的做法是，尽量不让拆分服务和数据结构的变化放在一起做，要尽量把一次大的重构划分为几次比较小的重构，但是在很多场景中，两者一起做还是无法避免的。

我们要保证的是“Zero Downtime”，也就是说，在任何时刻都不影响用户使用。一旦

发生故障，应该及时回滚，因此我们要同步数据，不但要保证老系统产生的数据同步到新系统，还要保证用户在新系统产生的数据能够同步到老系统。

在数据迁移的过程中经常采用以下两种方式。

#### 4.7.1 逻辑分离，物理不分离

逻辑分离，物理不分离（如图 4-15 所示）是指新服务和老服务放在一个数据库里，建立不同的表名，从代码层面实现隔离、解耦。数据迁移可以通过触发器实现，也可以采用双写的方式实现。通过单库事务实现业务服务写数据的同时写两张表。

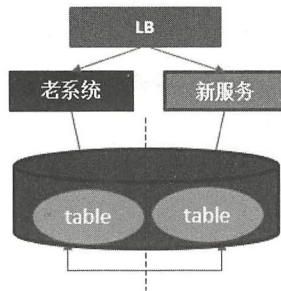


图 4-15 逻辑分离，物理不分离

逻辑分离，物理不分离的优点是足够简单，缺点是隔离性差，容易引发全局故障。

由于触发器导致数据库的扩展受到影响，因此这个方案更多地被作为一个临时方案或者过渡方案。

#### 4.7.2 物理分离

物理分离（如图 4-16 所示）是指新服务和老服务的数据通过不同的数据库物理隔离，可以使用相同的表名，数据同步需要额外的方案实现。

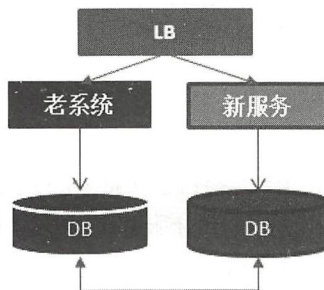


图 4-16 物理分离



物理分离的优点是隔离性好，缺点是数据同步比较复杂。

如果逻辑分离，物理不分离可以满足，则采用它，它更简单高效。如果采用物理分离，以下几种方案可以实现数据库同步。

- 利用数据库同步工具通过读取 binlog 实现数据双向同步。
- 在业务应用上同时双写两个数据库。
- 老系统在写数据库的同时，发送消息到消息中间件，消费消息从消息中间件实现同步。

无论采用以上哪种方式同步数据，都不可避免地会遇到一致性问题。

# 5

## 第5章 可扩展性设计

可扩展性的重要程度在很多系统中往往被低估。可扩展性是衡量系统架构优劣的一个非常重要的判断指标，但是不能盲目追求可扩展性，还要兼顾成本。以下示例将会告诉我们可扩展性设计的重要程度。

Facebook 在 2009 年每天产生 3 千万张照片；2013 年每天产生 3.5 亿张照片；2015 年每天产生 20 亿张照片。

阿里巴巴 2009 年首个“双 11”活动，一天内的销售额为 5 千万元；2012 年“双 11”活动一天内的销售额为 191 亿元；2017 年“双 11”活动一天内的销售额为 1682 亿元，而且是 11 秒内销售额破亿元。

看到这两组数据，我们不禁要问：仅仅通过增加机器就能解决问题吗？

### 5.1 加机器能解决问题吗

任何一个系统，随着业务的快速发展，都需要考虑或解决扩展性的问题。系统是否可以做到流量来临时通过扩展避免宕机，甚至不降低用户体验？很多互联网公司在用户规模增长速度比较快，技术积累不足的阶段都发生过宕机事件。比较典型的例子是聚美优品，它为店庆三周年促销活动做了大量宣传，受到了广泛关注。为了应对流量的爆发，聚美优品多次为服务器扩容，并制定了详细的技术应对方案，老板陈欧在微博中写道：“应对宕机出现，我们的后台已经准备好，如果真的出现宕机就给技术人员每人发一把刀切腹”。但是

当促销真正开始的时候，出现了大量宕机事故，如网页无法打开、永远都在排队中。用户戏称：“我终于明白了聚美优品三周年，‘打破低价，你想不到的折扣’这句广告语的真谛，就是无论价格多低我也进不去、看不见。”甚至有的用户认为宕机是阴谋，为了“少赔点”。总之，对于用户来说宕机是难以理解的。

另外一个例子，京东初期，和当当网、苏宁、国美频频开展“价格战”，京东商城 CEO 刘强东通过微博宣称：“京东商城所有大家电将在未来三年内保持零毛利，并保证比国美、苏宁连锁店便宜至少 10% 以上。”此话一出，苏宁立即开始反击，执行副总裁李斌通过微博宣称：“包括家电在内的所有产品价格必然低于京东。”活动开始不到 10 分钟，由于服务器瞬间流量暴增，苏宁官网和苏宁易购就出现访问困难和无法登录的情况，整个促销伴随的是服务器频频宕机、网站打不开。

从上面的两个例子能够看出，扩展性是多么重要，到了关键时刻绝不仅仅是加机器这么简单。

## 5.2 横向扩展

我们也可以把加机器得到的性能提升称为横向扩展。

横向扩展（scale out）也叫水平扩展，指用更多的节点支撑更大量的请求。例如如果 1 台机器支撑 10 000TPS，那么两台机器是否能支撑 20 000TPS？

纵向扩展（scale up）也叫垂直扩展，扩展一个点的能力支撑更大的请求，它通常通过提升硬件实现，例如把磁盘升级为 SSD。

横向扩展通常是为了提升吞吐量，响应时间一般要求不受吞吐量影响即可。因为本身在访问量比较小的时候，响应时间就是可接受的范围。例如去分布式缓存获取一条数据的响应时间在毫秒级，理想情况如图 5-1 所示，只要在吞吐量不断提升的情况下保持这个响应时间即可。当然，响应时间和吞吐量在资源一定的情况下，通常是互斥关系，如果要降低响应时间，可以通过纵向扩展提升单机能力，或者改变数据存储结构、压缩等方式。

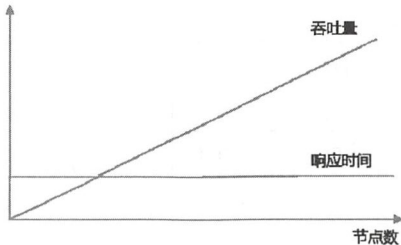


图 5-1 响应时间和吞吐量随节点数变化关系图



### 5.3 AKF 扩展立方体

提到可扩展性就不得不提著名的 AKF 扩展立方体（Scalability Cube）。AKF 是 eBay 前副总裁 Martin Abbott 在 *The Art of Scalability* 一书中提出的经典理论，他把系统在架构上的扩展性按照三个维度进行说明，如图 5-2 所示。

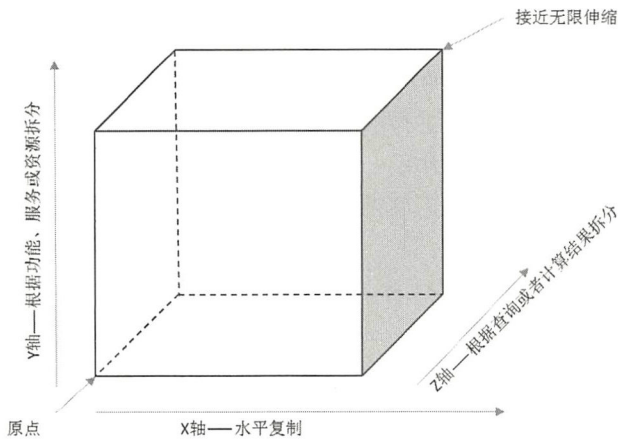


图 5-2 AKF 扩展立方体

下面我们通过表 5-1 简单说明一下三个轴适用的场景、优势及挑战。

表 5-1 AKF 扩展立方体

	描 述	场 景	优 势	挑 战
X 轴	通常称为水平扩展，通过复制实例，前端对流量进行负载均衡，以分摊整体压力为目的进行扩展	产品初期	架构简单，实施速度快，研发成本低。只需要通过负载均衡，保持无状态就非常容易达成	通常无状态的业务服务在产品初期可以通过 X 轴快速扩展，有状态的服务并不适用
Y 轴	根据服务或者资源扩展，也是我们从单体架构演进到微服务架构的思想	业务逻辑复杂，数据关联性不是特别强，团队规模大，代码规模大	故障隔离性好、快速部署、团队沟通效率高、容易实现业务复杂性分解等，可以参考微服务架构的优势	对工具环境依赖高、资源消耗多、运维复杂度高、一致性实现成本高等，可以参考微服务架构的挑战
Z 轴	根据查询或者计算结果进行拆分。简单来说，就是分片的思想，当数据规模非常大的时候，可以通过分片降低整体的压力	大型分布式系统；存在并发压力，X 轴、Y 轴扩展方式无法解决问题	可以基于分区扩展，扩展性更强，对于突破单张表的数据规模效果显著	架构复杂，数据迁移复杂，成本非常高

下面我们可以用一个例子来说明，系统是如何从初期一步步扩展的。假设我们现在要开发一个微博系统。

第一阶段，产品初期，可能只有十人左右的团队，还没有用户，需求也不确定。此时我们只要通过单体架构实现系统即可，如图 5-3 所示。为了容灾，可以在前端放置一个负载均衡服务，后端采用两个服务。数据库为了容灾也可以采用 Master-Slave 的架构。

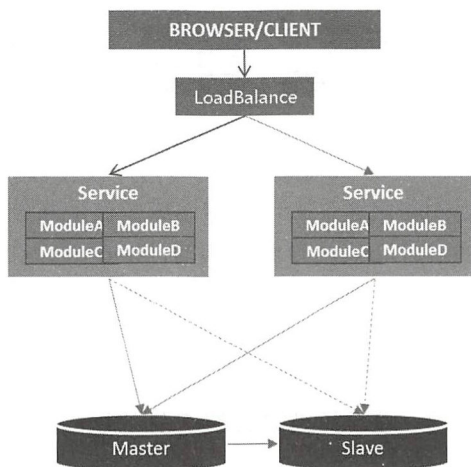


图 5-3 单体架构

随着用户访问量的快速增长，当系统出现性能瓶颈的时候，我们此时采用的是 X 轴的扩展方式，通过不断复制 Service，增加 Service 的数量来应对。当数据库出现瓶颈的时候，可以按照 X 轴扩展的思路做读写分离。

第二阶段，用户快速增长，产品需求快速增加，研发团队可能会接近百人，沟通效率越来越低，数据库主从延迟问题开始出现，磁盘压力逐步加大。为了缓解这些问题，首先作为过渡阶段，可以增加业务服务实例的数量，数据库可以通过提升硬件的性能暂时抵抗压力。另一方面，我们开始按照业务领域拆分服务，如图 5-4 所示，每个服务独享一个数据库，接口是服务与外部联系的唯一通道。这样既降低了耦合度，提高了沟通效率，又可以缓解数据库的压力，实现分库操作。但是在数据库中，单表的数据量持续增长，假设我们用 MySQL，随着数据量的增加，响应时间变长，吞吐量下降，此时我们首先会进行垂直分表。例如用户表有 100 个字段，但是常用的可能只是 10 个字段，我们会按照一对一的方式拆分为两张表，可以参考微服务架构拆分的相关内容。这就是按照 Y 轴扩展的思路。

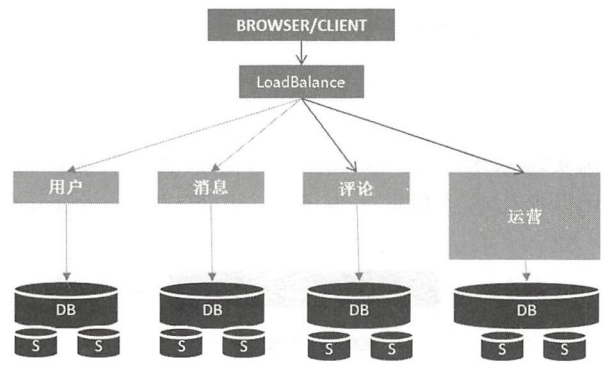


图 5-4 微服务架构

第三阶段，随着用户规模的快速扩大，数据库成了性能瓶颈。我们会通过 MQ 削峰填谷，解决写的性能瓶颈，通过分布式缓存解决读的性能瓶颈。在数据库一侧，我们会对表进行水平拆分，例如，有 3 千万条用户数据，可以根据用户 ID 拆分为 4 张表，每张表 750 万条数据。为了解决拆分带来的复杂性，可以通过数据库中间件屏蔽底层分表细节。当然，这时候很可能会遇到数据中心的容量瓶颈，促使我们去建立多数据中心，按照用户的地域切分数据，让数据离用户更近一些。这就是按照 Z 轴扩展的思路，如图 5-5 所示。

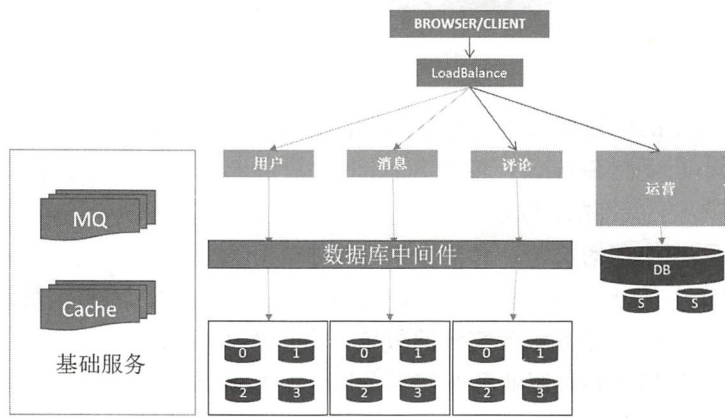


图 5-5 Z 轴扩展

## 5.4 如何扩展长连接

在微服务章节，我们已经了解到服务无状态的意义，但是有时候并不是所有的状态都可以外置，例如长连接。当长连接遇到负载均衡的时候就会变得非常复杂。



首先，应该尽量减少长连接，因为并不是每个应用都需要那么及时，大部分都可以通过短连接实现。例如电商的 App，所有的浏览、下单都可以基于短连接实现，极少需要长连接。

假设要设计一个微博 App 客户端和服务端的交互，如图 5-6 所示。当用户关注的人发了一条微博，此时如果用户不在线，并不需要马上收到，完全可以当用户上线的时候通过短连接去拉取消息。但是，如果用户正在看微博，就应该提醒他有几条新消息，这个通知是需要即时性的。

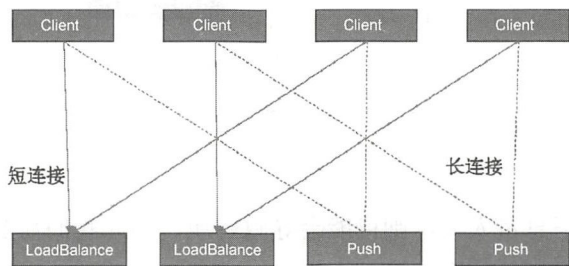


图 5-6 微博 App 客户端和服务端的交互

当然，即时消息也不一定非得是长连接，也可以采用定时轮询的方式实现。实际上，在真实的生产环境，为了保证到达率，通常需要长连接和短连接配合。因为长连接在网络条件不好的情况下，经常会出现各种各样的问题，导致消息不能及时到达。通过轮询心跳的方式可以定时拉取消息，减少长连接推送失败的情况。

那么，问题来了，当其中一个 Push 服务不可用的时候，客户端应该做何反应？

常用的方案有两种：反向代理和注册中心。

#### （1）反向代理

在 Client 和 Push 之间增加一层反向代理服务，如图 5-7 所示。Client 并不知道具体的 Push 服务，通过反向代理服务转发，另外反向代理也兼具负载均衡的作用。

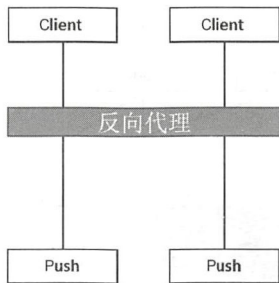


图 5-7 反向代理

## （2）注册中心

如果存在一个注册中心（如图 5-8 所示），那么 Client 在连接 Push 服务时可以先到注册中心获取可以连接的列表，然后根据 Push 服务的地址去连接 Push 服务，这样如果一个 Push 服务不可用，就可以让 Client 连到另外一个 Push 服务。

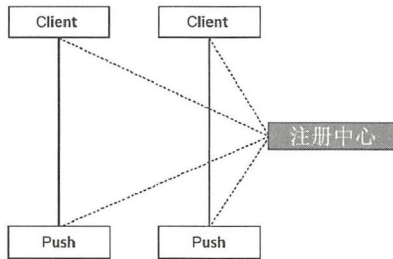


图 5-8 注册中心

如果需要推送的消息量太大，则可能会导致雪崩，因此我们需要限流，业界普遍会在推送服务前增加 MQ 以削峰填谷。

还有另外一个问题，当推送消息的时候，肯定要知道通过哪个 Push 服务去推送，因为 Client 不可能和所有的 Push 服务建立长连接，单机的连接数是有限的，如何记录 Client 和 Push 的对应关系呢？

方法一，记录 Client 和 PushServer 的映射关系，如图 5-9 所示，将关系数据放入缓存。

- 优势：扩容不影响现有节点连接。假设现在有两个节点扛不住压力了，扩容到 3 个节点，此时只需要把新的连接连到新的节点，已经存在的连接可以不用断开重连。
- 劣势：需要维护 Client 和 PushServer 的映射关系。

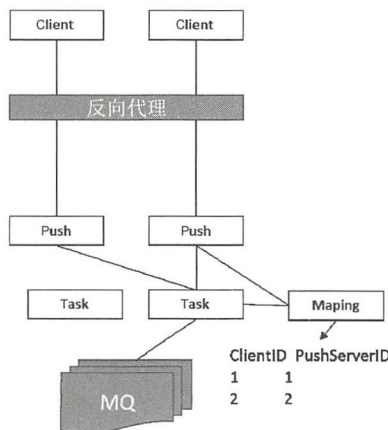


图 5-9 消息推送映射关系



方法二，根据节点总数计算映射关系。

- 优势：简单。只需要知道现在一共有几个 PushServer，根据 ClientID 对总节点求余、一致性哈希，或者根据用户 ID 的范围就应该知道推送到哪个节点。
- 劣势：首先，如果扩容需要断开一批连接，则重新连到新的 PushServer 节点有中断的可能；其次，如果根据范围计算，则有可能存在热点，即某个 PushServer 压力很大，而其他节点比较轻松；最后，如果一旦 Push 节点挂掉，就要重新建立连接。有时成本比较高，假设一共有四个节点，监控到其中一个节点故障时，要把这个节点的连接转移到其他三个节点上，一旦故障节点恢复，因为其他三个节点压力较大，我们又要让客户断开连接，连到新的节点。如果建立几个冗余节点，一旦发生故障，那么 back 节点迅速顶上来，而故障节点恢复后可以作为 back 节点，如图 5-10 所示。

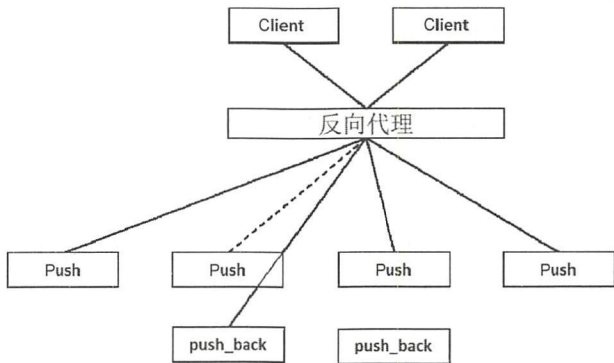


图 5-10 备份节点

## 5.5 如何扩展数据库

前面我们讲到了数据库扩展的流程，下面详细讲解每种扩展方案。

### 5.5.1 X 轴扩展——主从复制集群

假设 Service 访问数据库的吞吐量在 4 500TPS，其中写为 500TPS，这种典型的读多写少的场景通常会采用读写分离，将所有的读请求分发到 Slave 上，Master 只负责写，Master 和 Slave 之间通过数据库自带同步机制复制数据。Master-Slave 结构，如图 5-11 所示。





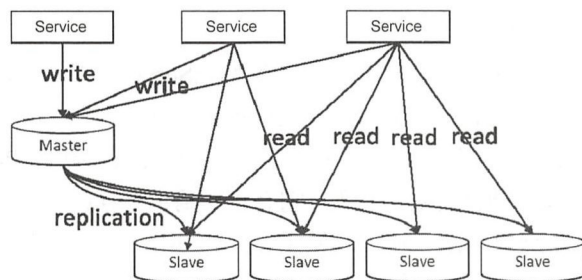


图 5-11 Master-Slave 结构

在这种方案中，多个 Slave 服务器异步复制 Master 的数据，复制步骤如下。

- (1) Master 将更新记录到二进制日志（binary log）中。
- (2) Slave 将 Master 的日志（binary log）复制到它的中继日志（relay log）中。
- (3) Slave 重做中继日志中的事件。

由于很多业务都符合读多写少的特点，使得这种扩展方式简单有效，可以很容易地缓解负载。采用主从复制还可以将服务进行隔离，例如，终端用户访问 Slave-01，系统内部统计工具访问 Slave-02，当运营人员做系统内部统计的时候会使 Slave-02 压力骤增，这种隔离方式将起到保护 Slave-01 不受影响的作用。

这个问题的问题在于当 Slave 增加到一定数量时，Slave 对 Master 的负载，以及网络带宽都会造成严重的影响。本身单库承受不了可能是因为磁盘 IO 达到上限，而同步数据同样需要消耗 Master 服务器的性能。

使用新版本有助于解决部分问题，如 MySQL 5.7 在主从复制方面提供了几个比较实用的功能。

- 多源复制（多主一从）。
- 半同步复制改进。
- 基于组提交（LOGICAL\_CLOCK）的并行复制。

### 5.5.2 Y 轴扩展——分库、垂直分表

Master-Slave 集群适合读多写少的场景，只能通过不断增加 Slave 实例个数来解决读的性能问题，只能通过 Master 写入，单节点的写入能力有限。况且，如果要满足写和读的一致性，就需要让读也访问 Master，当系统规模不断增大时，如果写成为了瓶颈点，就需要考虑 Y 轴的扩展了。首先要考虑的是分库，分库是指把原来一个数据库中的多张表根据数据量、访问量、关联程度分解到多个数据库中。通常分库操作是和微服务拆分同步进行的，可以根据微服务划分的原则进行划分，划分后每个服务独享一个数据库。分库的最大特点



就是相对简单，尤其适合各业务之间的耦合度比较低，业务逻辑非常清晰的系统。

分库相对于 Master-Slave 集群付出的成本更高，需要处理分布式事务问题、关联查询问题，但是相对于下面的方案更简单一些。

垂直分表是分库的一种特殊形式。有的业务中，单表字段数非常多，一些电商中的用户表可能超过 200 个字段。虽然表内的字段确实是一一对应的关系，但是实际上，并不是所有的字段都是常用的，通常这 200 个字段可能只有十几个字段是常用的。如果已经进行了分库，将用户表独立出来了，仍然存在性能问题，那么此时可以尝试垂直分表。也就是把单表的字段垂直拆分为多张表。初期可以放到一个数据库中，查询的时候更简单。如果仍然存在性能问题，可以分到不同的数据库，放在不同的物理机上。

### 5.5.3 Z 轴扩展——分片（sharding）

如果采用分库、垂直分表还是不能解决问题，那么就只能通过 Z 轴扩展的方式进行分片了。什么时候开始考虑分片呢？由于采用分片会导致架构的复杂度大幅上升，因此如果能避免应该尽量避免。一般按照经验值，MySQL 在单表十个字段以下，数据量达到 1 千万左右时，如果采用 SATA 磁盘，性能会遇到比较大的瓶颈。如果此时数据量还是大幅增长，就应该考虑分片了。

数据库分片的目标如下。

- 数据量尽可能分布均匀。因为数据量会对数据库造成压力，影响性能指标。在 100 条数据里搜索一条数据和在一亿条数据里搜索一条数据完全不一样。
- 访问量尽可能分布均匀。例如微博知名博主发布一条信息，可能会有几千万的转发量和评论量。最好不要存在某个点特别热，因为扩/缩容通常是整体架构的行为，当然也可以通过缓存的方式，让热点数据尽量命中缓存，缓解热点问题。
- 一次访问尽可能落到一个分片。在分片的时候，按照哪个 key 进行切分可以决定一次请求会访问几个分片。例如，当订单表按照订单 ID 进行切分，以买家维度进行查询时，势必造成要遍历所有的表，这样通过分片提升的性能就大打折扣，系统的扩展性受到了挑战。
- 数据迁移量尽可能少。当需要扩容的时候，数据迁移的过程是比较复杂的，为了不中断服务，需要迁移的数据量越少，对系统整体的压力就会越小。

数据库分片对原有的架构破坏性很大，需要考虑的地方很多，因此分片的算法至关重要，以下我们就来了解一下几种常用的分片算法。

#### （1）区间法（Range-Based）

区间法，如图 5-12 所示。假设一共有两千万条记录，此时我们可以按照 ID 的范围分



成四张表，每张表独占一个数据库。当然也可以根据时间、地域、组织进行切分，例如一个月一张表、一个省一张表、一个租户一张表等。电信级的应用很多基于省份分片，这样做的另一个好处是隔离性。

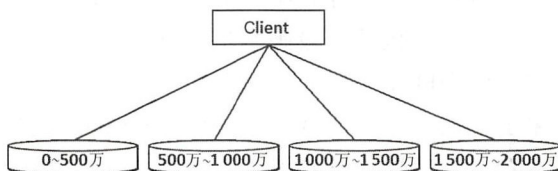


图 5-12 区间法

区间法的优点是利于排序，在这几种分区算法中，只有区间法配合分区算法更容易排序。区间法的缺点如下。

- 容易导致热点问题。假设图 5-12 中 500 万~1 000 万压力较大，此时应该如何分裂，如何迁移数据。
- 需要额外的元数据记录。

区间法的适用场景如下。

- 历史数据严重低于最近的数据访问，历史数据可以归档。例如，电商网站中订单的物流信息可能保留三个月就可以了。
- 数据分布相对比较均匀的场景。
- 数据按照区域需要隔离的场景。

## （2）轮流法（Round-Robin）

轮流法（如图 5-13 所示）根据关键字对分片总量求余以实现均匀分布。给定一个数据  $K$ ，应该放到哪个分区？我们可以用公式  $n = K \bmod N$  来计算放置的位置，其中  $N$  代表分片总数， $K$  代表分片关键字， $n$  就是我们要放的节点位置。如果把用户 ID 作为 Key， $\text{userID}=1$  的数据对 4 求余等于 1，应该放到第二个数据库中。如果 Key 是自增长的 int 或 long，则数据分布均匀，不容易出现热点问题。如果 Key 是规律的字母或数字组成的，则很容易出现问题，此时可以通过对 Key 计算 Hash 值缓解热点，公式变为  $n = \text{Hash}(K) \bmod N$ 。

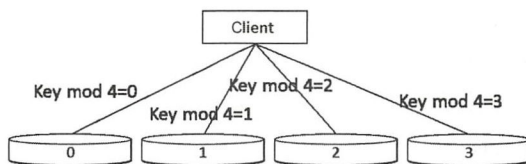


图 5-13 轮流法





采用轮流法进行扩容的时候，最好成倍扩容，这样迁移的数据量少。例如，从两个节点扩容为四个节点需要迁移一半的数据。我们举例说明一下，如果现在有两个库，采用轮流法分库，现在由两个库扩展为三个库，发生移动的数据量为三分之二。但是如果是从两个分片变为四个分片，只有一半的数据发生了移动，迁移的数据量更少，并且达成了扩容的效果。

轮流法的优点如下。

- 简单，开发运维人员看到 Key 的时候，很容易知道这条数据应该在哪个分片。
- 不需要维护元数据。

轮流法的缺点如下。

- 当进行扩/缩容的时候，迁移的数据量较大。
- 不容易排序。

轮流法的适用场景如下。

- 不经常扩/缩容的场景。
- 不需要排序或者可以用其他方式代替的场景。

### (3) 一致性哈希法 (Consistent Hash)

假设把一张用户表水平切分为两张表，用户 ID 是自增长的。现在计算一下用户 ID 是 1、2、3、4、5、6、7、8、9 的数据应该如何分布。按照轮流法进行 MOD，结果如图 5-14 所示，DB-0 存储 2/4/6/8，DB-1 存储 1/3/5/7/9。

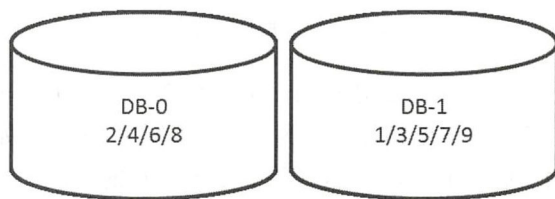


图 5-14 轮流法

如果现在进行扩容，从两个数据库扩展到 3 个数据库，那么 DB-0 中的 2 要迁移到 DB-2 中，4 要迁移到 DB-1 中；DB-1 中的 3 要迁移到 DB-0 中，DB-1 中的 5 要迁移到 DB-2 中，只剩下方框内的数据没有发生移动，如图 5-15 所示。实际上，DB-2 只存储了 3 条数据，却移动了 5 条数据（所以，一般基于 MOD 算法的数据扩容，通常是基于倍数进行的，就是为了减少数据迁移量）。



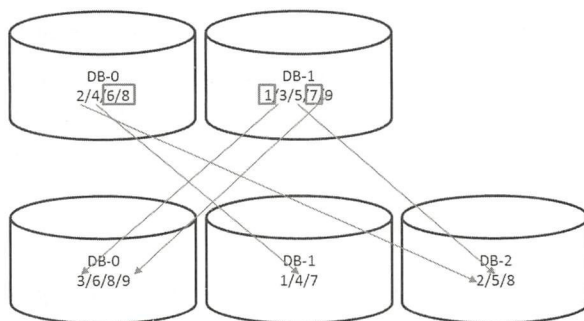


图 5-15 数据迁移

那么有没有办法只移动 3 条数据呢？

一致性哈希法就是为了解决这个问题而生的。一致性哈希法是 1997 年由麻省理工学院提出的一种分布式哈希（DHT）实现算法，设计目标是解决因特网中的热点（Hot Spot）问题，一致性哈希法相比其他算法可以减少数据的迁移量。一致性哈希法的架构，如图 5-16 所示。

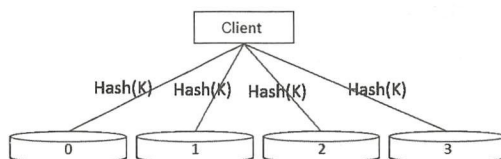


图 5-16 一致性哈希法

首先将 Key 按照常用的哈希算法对应到一个具有  $2^{32}$  个桶的空间中，即  $0 \sim 2^{32}-1$  的数字空间中。我们可以将这些数字头尾相连，想象成一个闭合的环形，如图 5-17 所示。 $2^{32}$  是 42 亿，这相当于有了 42 亿个节点，当然这些节点不必真的对应一个数据库，可以认为是一个虚拟节点。

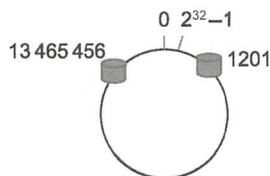


图 5-17 哈希环（一）

现在假设将两个数据库的 IP、HostName 加上端口计算出一个 hash code 值，如果 DB-0 是 1 201（虚拟的），DB-1 是 13 465 456，将这两个节点分布在这个环上，那么所有的数据



如果通过 hash code 后取模计算出的结果落在 0~1 201 内,就放到 DB-0 上,如果在 1 201~13 465 456 内或者大于 13 465 456 就放到 DB-1 上,如图 5-18 所示。一致性哈希是按照顺时针方向分布数据的。

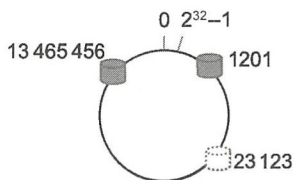


图 5-18 哈希环 (二)

继续用上面的例子,将用户 ID (1~9) 也用同样的方法算出 hash code,并对 42 亿取模将其存放到环形节点上。假设 1/4/6/7 落在了 DB-0 上,2/3/5/8/9 落在了 DB-1 上,如果现在新增一个节点,假设按照 IP、HostName 加上端口计算出一个 hash code 值是 23 123,那么只有落在 DB-1 上的数据 (2/3/5/8/9) 涉及迁移,DB-0 上的数据可以保持不变,可能 3/8 迁移到了新的节点。

我们简单介绍了一致性哈希法的大致原理,到这里大家应该大致明白一致性哈希法迁移的数据量比较小,是因为一致性哈希最终是基于范围迁移的。相对于直接通过范围分片,一致性哈希做了一次哈希值计算,分散了热点。当然,这里存在一个比较大的问题,当节点比较少的时候,数据分布不均会导致热点产生,为了解决这个问题,可以引入虚拟节点 (virtual node) 的机制。

如图 5-19 所示,可以针对每个节点虚拟出  $N$  个节点,因为虚拟出的节点是按照 hash code 对 42 亿取模结果放到哈希环上的,所以不会排好序,它们是散落在环上的,也就是说 DB-0 对应的是 DB-0-1 和 DB-0-2。它们在环上并不是一定会挨在一起的,当虚拟节点足够多的时候,它们是平均散落在环上的。

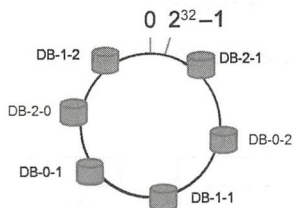


图 5-19 哈希环 (三)

假设现在 DB-0 发生故障,按照顺时针,DB-0-1 的数据会落到 DB-2-0 上,DB-0-2 的数据会落在 DB-1-1 上。



一致性哈希法的优点是扩/缩容数据迁移量较少。它的缺点有两个，一是算法复杂，不容易调试；二是不容易排序。

一致性哈希的适用于不需要排序或者可以用其他方式代替的场景。

当采用水平切分的时候，可能会遇到很多问题，以下两点尤其需要注意。

如何避免重新均衡数据？因为重新均衡是昂贵的。需要重新均衡的原因是分片是静态的，但是数据和业务都是动态的。也就是说，可能现在是均衡的，过了一段时间变成不均衡的了；也有可能某段时间是均衡的，另外一段时间是不均衡的。如很多 SNS 类的网站，初始阶段活跃用户数可能在 1 千万以下，过了几年，活跃用户数可能会集中在 1 亿到 2 亿之间，这是一个动态变化的过程。因此分片时要用发展的眼光看问题。

管理分片是一个复杂的问题。我们很难实现数据的强一致性，如果采用分布式事务，会使性能、扩展性受到很大影响。如何进行分页、排序等查询？以前一条 SQL 就搞定了，如果使用了分片，当进行分页、排序的时候，就会变得非常复杂。例如为了避免热点，根据哈希分了 64 张表，查询出按时间排序后的第 10 000 条到 100 010 条数据，这就需要去所有分片取数据，然后在内存中进行计算。当然，也可以建立另一维度的数据去解决，可以参考 5.5.5 节，但是这增加了架构的复杂度，有可能还要为此引入其他的服务。因此，水平分表应该作为最后一个选择。

#### 5.5.4 为什么要带拆分键

我们都知道当数据库进行水平分表的时候，需要通过拆分键路由进行查询，这是为什么呢？因为如果不带拆分键，就要到所有的表中去查询，数据库中间件不知道去哪查。虽然可以通过并行的方式查询所有表，但是这会导致数据库的压力增大。

是否带拆分键的对比，如图 5-20 所示，如果带上拆分键 uid，则很容易定位到 DB2；如果不带拆分键 uid，则数据库中间件不知道去哪查，只能查询所有的数据库。

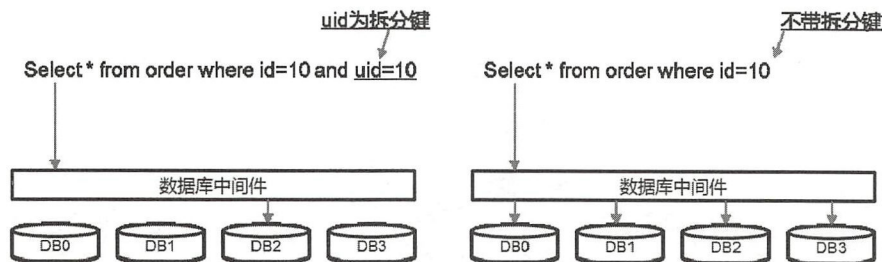


图 5-20 是否带拆分键对比

### 5.5.5 分片后的关联查询问题

我们通过一个例子来了解这个问题。例如，一个库中包含一张用户表和一张订单表，假如查询“北京的订单金额大于 100 的数量”，在一个库中可以通过关联查询完成。如果用户和订单被划分到了不同的服务，再进行关联查询，就非常麻烦了。

#### 方案一：建立多维度数据库

可以尝试建立另外一个综合数据库，相当于为了进行关联查询多冗余了一份数据。

建立多维度数据库是电商中比较典型的例子，如图 5-21 所示。实施微服务架构后，商品、价格、库存垂直划分为多个独立的数据库。

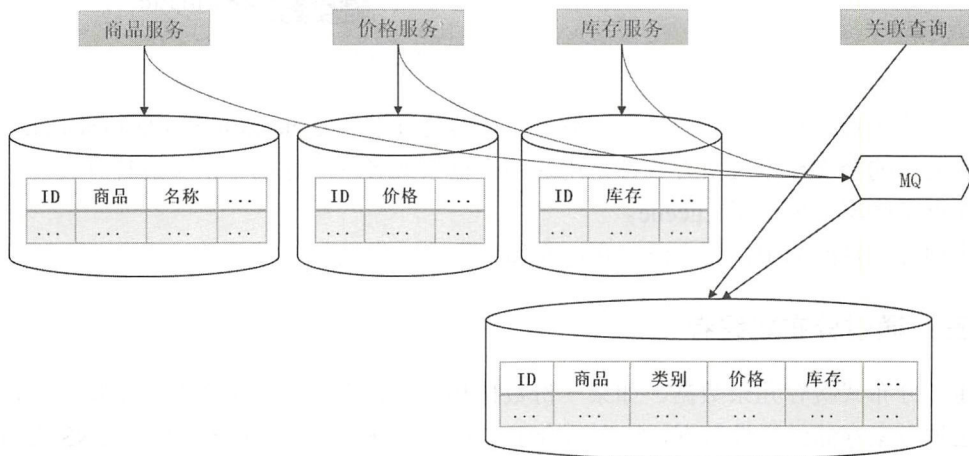


图 5-21 多维度数据库

更新时，通过消息中间件异步更新到综合数据库内。

查询时，直接从综合数据库查询。

虽然综合表有可能变得臃肿，但是综合表的查询一般是后台管理人员使用的，查询频率较低。当然，综合表可以替换为 MongoDB，因为 MongoDB 可以自动伸缩，运维的工作量要更少。

一个类似的做法是在大数据平台建立综合数据查询系统，问题是有可能存在延迟，需要根据具体业务场景决定。

建立多维度数据库方案的优势是架构简单，而它的问题主要包含如下两方面。

- 可能存在不一致的风险。
- 综合表有可能变得庞大无比，如果查询量比较大，则可能会成为性能瓶颈。

## 方案二：建立外部搜索引擎

在图 5-22 中，通过分布式的搜索引擎，建立倒排索引，进行全文检索。

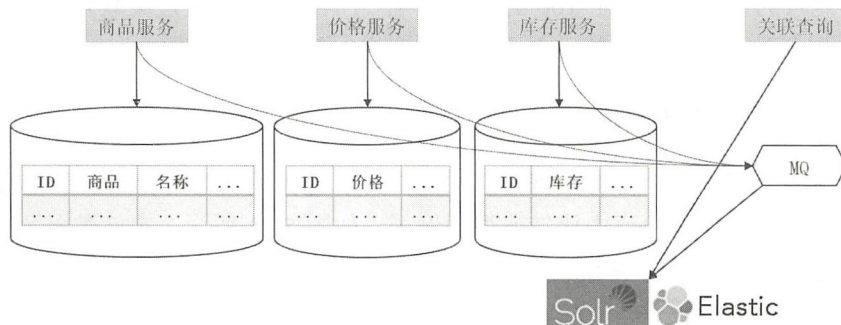


图 5-22 建立外部搜索引擎

全文检索目前有很多开源方案，最为流行的莫过于 Apache Solr 和 Elasticsearch。很多互联网公司的全文检索解决方案都是用这两个框架或者基于这两个框架进行开发的。它们的共同点是底层都采用了 Lucene。

这种方案目前应用比较广泛，在电商场景中比较常见。

## 方案三：通过分布式缓存

通过分布式缓存冗余数据。如果一份数据相对比较小，占用空间并不是特别大，则可以用这种方案存储结果性数据，特别适合于多对多的场景。这种方案常用在 SNS 类系统的综合查询中。

### 5.5.6 分片扩容 (re-sharding)

分片的时候，应该从长期考虑，避免频繁地进行重新分片，因为重新分片会导致大量的数据迁移。可以根据未来数据量的增长速度、架构调整的可能性进行规划，因为预留太多会导致成本增加，而预留太少又会导致频繁迁移。根据经验值可以预留出未来 1~2 年的空间，如果害怕资源浪费，则可以把多个实例部署到一台服务器上。

假设现在有两个分片采用 MOD 的分片算法，如果存在 0~9 的用户 ID，进行 MOD 后的分布情况是 DB-0 (0/2/4/6/8)、DB-1 (1/3/5/7/9)。按照倍数扩容迁移的比例最小，因为当数据量较大的时候，迁移会对系统造成很大压力，应该尽量减少迁移。

以 MySQL 为例，应该选择流量较小的时段进行扩容，禁止在高峰期扩容。



### 方案一：停服扩容

停服扩容（如图 5-23 所示）的步骤如下。

- （1）选择好升级的时间段，评估升级所需时长，对外沟通，挂出公告。
- （2）到时间后，所有流量在前端负载均衡处转发到停服公告页面，观察数据库状态，没有流量后先对数据库进行备份，然后开始升级。
- （3）新建两个数据库，分别命名为 DB-2、DB-3。可以通过一些开源的迁移工具，也可以自己开发一个迁移服务，或者利用存储过程进行迁移数据。
- （4）迁移完成后，删除 DB-0、DB-1 冗余的数据，验证数据完整性和一致性。
- （5）如果有数据库中间件，则修改中间件分片策略；如果没有数据库中间件，则修改 Service 的分片策略。
- （6）验证。如果没有问题则流量切回；如果发现问题，则挂出公告利用前面的备份进行回滚。

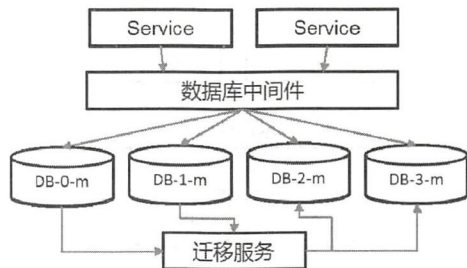


图 5-23 停服扩容

这种做法比较简单，容易操作，但是需要停止服务，要求一次性做对，否则回滚比较麻烦。这在产品初期，访问量不是特别高，技术实力较差时可以选择停服扩容。通常需要扩容的情况，一般数据量都比较大了。技术人员的水平、熟练程度、前期的准备工作对于这种方案的成败起到了决定性的作用，包括中断的时长也会受到以上因素影响。

### 方案二：基于数据库的 0 中断扩容

0 中断并非不能有中断，而是中断的时间足够短，可以忽略，不需要通知用户，或者对用户体验影响不大。以下案例假设存在数据库中间件，如果没有，则需要基于业务服务进行操作。

- （1）选择好升级的时间段，先备份数据库。
- （2）通常为了容灾提升读性能，此时应该已经有了 Slave 节点。如果没有 Slave，则需

要先分别为两个数据库建立 Slave 实例 DB-2、DB-3，如图 5-24 所示。

(3) 当 Master 和 Slave 之间延迟较小时，修改数据库中间件配置，停止写入，只能读取。将 Slave 提升为 Master，如图 5-25 所示。

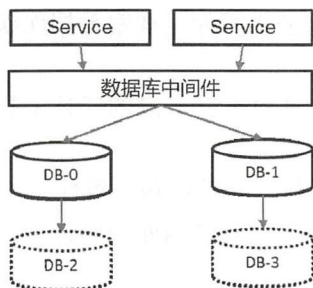


图 5-24 建立 Slave

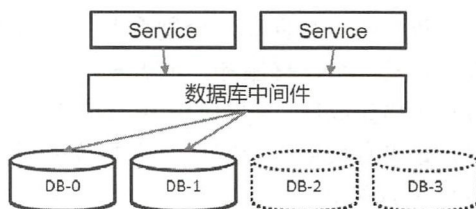


图 5-25 将 Slave 提升为 Master

(4) 修改数据库中间件配置，分片规则改为四个库，如图 5-26 所示。

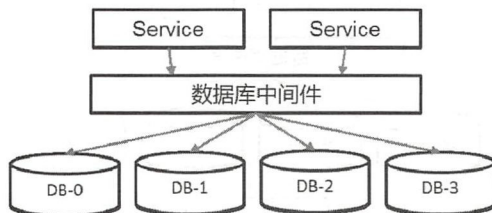


图 5-26 修改数据库中间件配置

(5) 修改数据库中间件配置，允许正常读写。注意，此时存在冗余数据，必须要求数据库中间件具备排重功能。

(6) 删除冗余数据。

注意，从 (3) 到 (5) 读取不受影响。写入是中断的，中断的时长取决于前期的准备工作和操作的熟练程度，一般可以控制到分钟级。如果希望中断时间更短，则可以把主从同步修改为主主同步或者半同步复制，但是这需要提前做好准备。

另外，也可以把所有的写入数据暂时记入日志或者写入 MQ，等到主从完全一致之后，先写入日志或者 MQ 的数据。

### 方案三：基于数据库中间件的 0 中断扩容

基于数据库中间件的 0 中断扩容方案和方案二类似，只不过它是通过数据库中间件完成的，大致步骤如下。

- (1) 备份数据。
- (2) 建立另外两个数据库实例 DB-2、DB-3，分别作为 DB-0、DB-1 的副本。可以通过迁移工具（基于状态机模式读取 binlog）让数据尽量接近。
- (3) 修改数据库中间件配置，停止 update 和 delete，只能 insert 和读取。insert 的时候需要通过数据库中间件进行双写，将 DB-0 的数据同步写入 DB-2 中，将 DB-1 的数据同步写入 DB-3 中。
- (4) 比对数据，当 DB-0 和 DB-2 的数据完全一致，DB-1 和 DB-3 的数据也完全一致时，修改数据库中间件配置，开始 update 和 delete。
- (5) 修改数据库中间件配置，修改分片规则，切换为 4 个库进行读写。
- (6) 删除冗余数据，扩容结束，如图 5-27 所示。

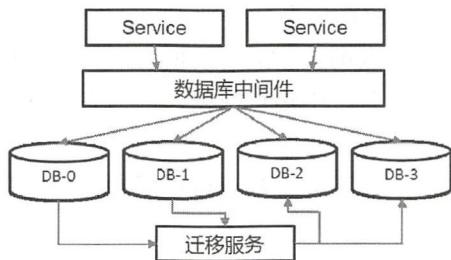


图 5-27 基于数据库中间件的 0 中断扩容

这个方案中断的只有 update 和 delete，如果业务场景中这两个操作很少，那么比较适合这个方案。

在实际的业务中，如果复杂度较高，则会混合使用以上分片算法。例如微信红包的分片规则为 db\_xx.t\_y\_dd，xx 和 y 代表红包 ID 的 hash 值后三位，dd 代表天数，混用了一致性哈希法和区间法。通过这种混合的分片算法，既避免了区间法导致的热点数据的问题，又有利于迁移数据，可以按照天为单位进行整张表迁移。

### 5.5.7 精选案例

#### 案例一：活动平台数据表水平切分

假设现在有一个活动平台，管理员可以创建活动，为活动添加用户，针对一个活动给用户发送促销短信或邮件提醒，如图 5-28 所示。按照场景，正常的操作逻辑是，查询关注某活动的所有用户，并且发送消息。





图 5-28 活动平台需求关系

这是一个典型的多对多关系。通常当用户数据量比较大的时候，会先选择分库，也就是把用户表单独放在一个数据库中，活动和活动用户关系表放到另一个数据库中。活动一般不会特别多，或者说活跃的活动一般不会特别多，但是活动用户关系表会非常大。

如果活动库变得非常大，那么此时该如何切分？一般不会选择直接把活动用户关系表单独放到一个库中，由于活动的数据量不是特别大，分库的效果不好，避免不了水平分表。

如果按照活动进行分片，当查询关注活动的所有用户时，则可以在一个分片内查出所有数据，但是按照活动进行分片会产生热点数据，因为有的活动可能用户比较多，而有的活动用户比较少。

如果按照用户进行切分，虽然不会有热点数据问题，但是当查询关注活动的所有用户时，需要遍历所有分片。

实际上，关系数据只是两个 id 而已，数据条数可能比较多，但是占用的存储空间并不会特别大，可以优先考虑通过缓存缓解数据库的压力，不做分区。活动有明显的时效性，一旦活动结束，数据就可以归档到历史数据库了。因为关系数据一般不会更新，可以将缓存的过期时间设置长一点。

## 案例二：SNS 数据表水平切分

很多人用过微博、微信，它们主要的表结构包括用户表、用户关系表（谁关注了谁）、消息表（发的微博、朋友圈）。SNS 需求关系，如图 5-29 所示。用户之间是有关系的，通常发布的内容按照用户的查询方式有多个维度。例如在微博中，首页通常是 timeline<sup>①</sup>，它是用户自己看的比较多的页面，还有一个页面是 profile<sup>②</sup>，关注者会经常访问。当消息的量比较大时，需要进行水平分表，如何切分保证不去遍历所有的分片呢？

① timeline 是指你可以看到的你关注的所有人发布的消息流，通常按照时间排序形成。

② profile 是指自己或者单个人发布的所有信息的流，通常按照时间排序形成。

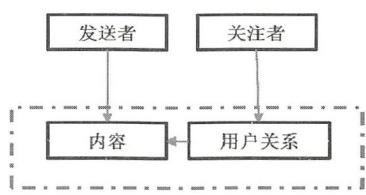


图 5-29 SNS 需求关系

如果按照发布消息的用户 ID 去切分数据，在查询 profile 的时候，可以在一个分片取到所有数据，但是查看 timeline 的时候，就要遍历所有的分片数据。

根据需求有两张表，分别是消息表（如表 5-2、表 5-3 所示）和用户关注表（如表 5-4 所示）。如果消息表被分为 8 张表，根据求余的算法，用户 ID 为 1 的消息会落入第一个分片，用户 ID 为 2 的消息会落入第二个分片。如果查询用户 ID 为 1 的 profile 页面，则可以直接在第一个分片取到所有数据。但是在用户关注表中我们不难发现，用户 ID 为 10001 的用户关注了用户 ID 为 1 和 2 的两个用户。如果用户 10001 查询自己的 timeline 页面，需要遍历所有分片，而 timeline 才是读取量比较大的页面，这样性能会非常低。

表 5-2 消息表分片 1

消息 ID	发布消息的用户 ID	消息内容
1000	1	.....
1001	1	.....

表 5-3 消息表分片 2

消息 ID	发布消息的用户 ID	消息内容
1005	2	.....
1006	2	.....

表 5-4 用户关注表

用户 ID	关注的用户 ID
10001	1
10001	2

提升读性能的关键方案之一就是冗余，除了上面的消息内容表，还可以再增加一张表，让 timeline 的数据在一个分片内取到，一次兼顾两个维度的查询性能。它带来的问题是可能会比较浪费资源。这也是目前 Twitter 采用的方案，由于国内的 SNS 存在大量“僵尸”用户，一般都采用推拉结合的方式兼顾。

### 案例三：电商数据表水平切分

电商中以订单为典型，一个订单包含订单 ID、买家 ID、卖家 ID 三个比较重要的查询关键字。为了简化，我们先不考虑订单和子订单相关的内容。

那么该如何选择水平切分键呢？

如果按照订单 ID 切分，则按照卖家 ID 和买家 ID 查询会遍历所有的表。

如果按照买家 ID 切分，则按照订单 ID 和卖家 ID 查询会遍历所有的表。

如果按照卖家 ID 切分，则按照订单 ID 和买家 ID 查询会遍历所有的表。

现在我们先来分析一下业务场景。

- 80%以上的查询是通过订单 ID 进行的。
- 15%左右的查询是通过买家 ID 来查询已购买商品列表的。
- 5%左右的查询是通过卖家 ID 来查询已卖出商品的列表的。

各个电商平台业务场景不一，因此以上数据的比例会有所浮动。

很明显，根据业务场景，如果只能选一个，那么最好选择订单 ID 进行切分。如果按照卖家 ID 和买家 ID 查询如何解决呢？

第一种方法，我们可以为卖家和买家分别存储一个维度的冗余数据，以供查询，但是这样就是三倍冗余，比较浪费。

第二种方法是通过外置搜索引擎建立索引进行查询。它比上一种方法好，因为用户买家 ID 或者卖家 ID 进行查询的时候，通常还有其他过滤条件。

还有一种比较讨巧的做法就是，让订单 ID 和买家 ID 建立联系。我们在水平切分表的时候，可以截取买家 ID 的后几位加在生成订单 ID 的末尾。如果在生成订单 ID 的时候能够保持后几位和买家 ID 的后几位一致，那么订单 ID 和买家 ID 就相当于建立了联系。通过订单 ID 和买家 ID 查询都不用遍历所有表，具体做法可以参考第 3.9 节关于分布式 ID 生成方案的介绍。

## 5.6 如何扩展数据中心

### 5.6.1 两地三中心和同城多活

为了容灾，大型企业系统通常会采用灾备的部署方式，也就是说，一个主机房、一个备份机房。主机房承载所有的流量，而备份机房平时处于休眠状态，没有流量，只有当发生灾难，主机房不可用的时候，才会启用备份机房。通常主备机房之间的延迟比较大，只能采用异步复制数据的方式。但是这仅仅作为灾备，如果一个机房的容量达到瓶颈，是无



法扩展的。

为了解决这个问题，通常会采用两地三中心的方案，这也是很多银行系统使用的方案。两地三中心的方案，如图 5-30 所示。其中两个机房需要离得比较近，延迟非常小，某些场景下可以看成是一个机房，所以需要专线连接，所有的流量都由这两个机房承载，另外一个则作为灾备，平时处于休眠状态。

两地三中心的方案有如下问题。

- 由于需要低延迟，专线比较贵，“两地”通常离得比较近，真正的灾难来临，例如地震，备份机房还是跟主机房一样受到破坏。
- 备份节点处于休眠状态，比较浪费资源，而且真正切换的时候，需要的时间较长。

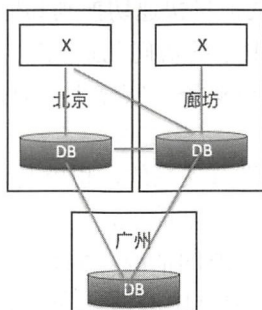


图 5-30 两地三中心的方案

我们需要明确的是，在两地三中心这种数据中心架构中，首先需要梳理出核心业务流程，对服务进行分级，规划哪些业务需要部署到多个数据中心，并不是所有的应用都需要做到那么高的可用性。

其次，往备份节点同步数据一定是异步的复制方式，否则性能会受到非常大的影响。异步意味着可能会丢失部分数据，这是不可避免的。如果真的发生地震，只要磁盘没被损坏，那么还是可以恢复的，尽管恢复的时间比较长。

### 5.6.2 同城多活

微信采用的是“一地三园区”的部署方式，三个园区是对等并且物理隔离的。也就是说，无论是服务层，还是存储层，都部署在三个园区，流量被负载均衡到三个园区，不存在资源浪费。当其中一个园区发生故障时，流量会被均分到另外两个园区，它们的流量分别上升百分之五十。这是由微信的底层存储决定的，微信底层存储采用的是 KVSvr，可以实现强一致、高可用、高性能。实际上，它利用了  $W+R>N$  原理，只要写数据时写两份成功就返回，它能保证如果其中的一个节点不可用，读取另两个节点一定能读到一份最新的

数据。当然，“一地”决定了时延较低，但是也无法做到跨城区的容灾。

我们来总结一下上面的方案。

- 如果跨城区，能跨城市容灾，但是成本高、利用率低、一致性低。
- 如果同城，成本低、利用率高，可以保证核心业务强一致，但不能跨城市容灾，且扩展性受限。

那么有没有其他的方案呢？当然有，不过，系统架构将变得越来越复杂，成本越来越高。

### 5.6.3 异地多活

异地多活是指在两个以上的城市建立机房，流量被平均分配，当一个城市的机房出现故障的时候，可以将发生故障的城市的流量快速切换到其他城市的机房中。异地多活就像以城市为单位的分片，例如，IP 地址为北京的访问北京机房，IP 地址为上海的访问上海机房。

如果做到异地多活，首先要分析业务，实际上，有的业务要实现多活是比较容易的，而有的业务是很难实现多活的。

例如手机通讯录，用户之间是没有关系的，也就是说 A 的通讯录里面存的数据不需要和其他人有任何关系。这种数据结构非常简单，可以以用户所在城市为切分键，这样就不存在一行数据在多个数据中心同时修改的情况。假设将中国分为三个数据中心，分别为北京、上海、广州，当用户属于北京数据中心时，所有请求都访问北京。新增数据时，只要北京写入成功就返回，然后同步组件订阅数据库 binlog 信息到其他机房汇总、备份，如图 5-31 所示。

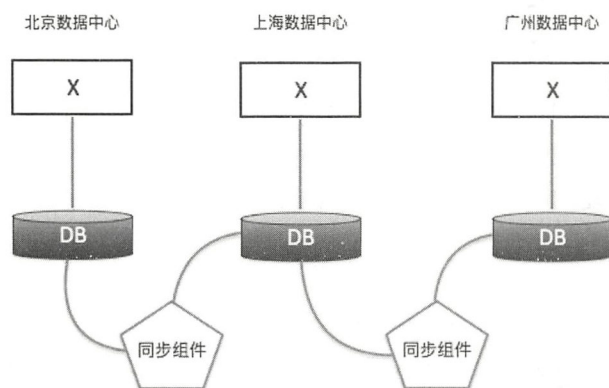


图 5-31 异地多活

如果是微博、电商这种数据，选择切分键就比较麻烦。在电商中，有买家和卖家两个维度读写数据，当买家下订单后，卖家维度要及时扣减库存，生成订单。而买家和卖家很

可能不在一个城市。由于一般电商都是微服务架构，一次下单操作，在后台可能是几百次请求调用，如果这些请求要在多个数据中心来回传递 10 次，假设每次跨数据中心要延迟 50 毫秒，那一次下单操作在采用异地多活架构之后就要比以前响应时间减少 500 毫秒，这是不能接受的。所以，我们最好能保证一次请求可以在一个机房内完成。

由于异地多活的成本比较高，不只是物理成本，还有设计、开发、维护的成本，因此，应该尽量让核心业务实现异地多活，而不是所有业务一视同仁。这个思路和数据库扩展的思路是一致的。如果能将一个独立的业务拆分到另外一个机房，则优先选择这种方案；如果依赖太多，解决不了，则选择异地多活。另外一个原因是有些业务很难实现拆分，特别是一些对一致性要求特别高的服务，比如库存，异步将是致命的。

在考虑异地多活的时候，还要考虑一致性问题，前面提到了一行数据的一致性问题，如果是整张表的一致性呢？如注册手机号，需要唯一键约束，如果分布到多张表中，无法实现一致性约束，虽然前端在写入前做了很充分的校验，但是如果一个用户进入多个注册页面，填写了校验信息，瞬间提交所有注册请求，由于异步的问题，可能导致全部成功。这时候有两种解决方案，一种是从业务的角度出发，这种毕竟是少数，如果前端已经做了防重复提交，那就可能存在恶意行为，应该由另外的定时任务去补偿处理。另外一种从技术角度出发，如果认为业务不允许出现任何不一致，那么此业务可以不做异地多活，可以做到多个数据中心读取，一个数据中心写入。



# 6

## 第6章 性能设计

性能优化是任何架构都不能跳过的步骤，贯穿系统的整个生命周期，而在微服务架构中，由于服务的划分，导致原本一次请求变成几次或者几十次，性能严重下降。在服务划分架构设计阶段，性能问题往往容易被忽略。直到压力测试或发布阶段，甚至是随着规模越来越大，性能问题才会逐步显现。性能下降可能会意味着要通过增加资源的方式来弥补，这会造成成本的上升，但是并不是所有的性能问题都可以通过增加资源解决。性能问题应该在架构设计阶段就开始考虑，贯穿整个开发过程，需要根据压力测试结果和生产环境的运行情况逐步优化。

比较常见的性能问题如下。

- 内存泄漏——导致内存耗尽。
- 过载——突发流量，大量超时重试。
- 网络瓶颈——需要加载的内容太多。
- 阻塞——无尽的等待。
- 锁——通过限制。
- IO 繁忙——大量的读写、分布式。
- CPU 繁忙——计算型常见问题。
- 长请求阻塞——连接耗尽。

## 6.1 性能指标

性能到底表现在哪些方面，通过哪些指标来体现呢？谈论性能的时候通常会提起以下两个指标。

- 响应时间（Latency），就是发送请求和返回结果的耗时。
- 吞吐量（Throughput），就是单位时间内的响应次数。

当然这两个指标在一定资源限制下才有意义，也就是占用几台服务器。另外，在资源一定的情况下，往往吞吐量和响应时间之间互相影响。你很难通过响应时间算出吞吐量，也不能通过吞吐量算出响应时间，需要单独通过压力测试得出结果。在资源一定的情况下，性能优化的本质就是榨取资源，利用一切可以利用的资源。例如，当 CPU 消耗较少的时候，可以考虑增加线程榨取 CPU 的能力。

除此之外，还有几个重要的指标也经常在性能优化阶段被提起。

- 负载敏感度，是指响应时间随时间变化的程度。例如，当用户增加时，系统响应时间的衰减速度。
- 可伸缩性，是指向系统增加资源对性能的影响。例如，要使吞吐量增加一倍，需要增加多少服务器。

## 6.2 如何树立目标

通常你得到的需求目标是这样的：“速度要快！”“跟以前比，性能不能降低”。很明显，这并不是一个有效的目标。

下面列出几种常见却无效的目标，并逐一分析它们无效的原因。

**平均响应时间 1s。**看这组数据，[2, 5, 3, 4, 301, 4, 2, 8, 7, 3, 3, 1, 1, 8, 2]  $AVG(f)=23.6$ ，平均响应时间因为一次非常严重的超时导致偏离，平均响应时间无法正确描述响应时间的状况。

**99%的请求要在 1s 内完成。**首先，不同的业务响应时间不应该相同，如提交一个订单和浏览一个详情页的响应时间差距会比较明显。其次，单纯的设定响应时间毫无意义，要在吞吐量下进行设定。

**支持 100 万用户。**首先，这并不等于系统吞吐量的设定，100 万用户有多少活跃用户，用户的操作频率如何。其次，增长速度是系统目前有多大数据量的前提，如系统没有数据和单表已经超过 1 亿条数据差别很大，最好能够同步设定。

**错误率。**指标再高，有错误也没用。

如何设立目标呢？我们可以尝试这样来设定，例如我们设定如下指标来描述一个电商中的订单列表。

- 响应时间：99% (TP99) < 1s。
- 吞吐量大于 10 万 TPS。
- 系统数据量：10 亿 > 订单表 > 1 亿。
- 数据增长速度：1 亿/年。
- 资源限制：几台服务器、什么配置、网络环境等。

同时，其他目标也要同步设定，例如系统整体可用性、一致性等，需要结合业务场景进行优化。在工作中，经常见到这样的情况：公司高层要求优化性能，而开发人员虽然优化了某个指标，但是导致了其他指标下降，成本升高。性能优化不应该从公司高层发起，而应该由技术人员定义、执行。但是，技术人员也不能为了满足成就感，而去过度优化。如在数据库性能达到瓶颈期，数据量增长不大的情况下，完全可以通过提升硬件解决问题，但是技术人员往往选择水平分表，导致架构复杂度大幅提升。

## 6.3 如何寻找平衡点

在优化的过程中，可能通过异步的方式增加了吞吐量，但是同时却降低了响应时间。因为提升吞吐量使系统负载增加，如 CPU 使用率、磁盘 IO 等，而负载的增加一定程度上降低了响应时间。那我们在做性能优化的时候应该如何抉择呢？

我们需要找到那个平衡点（即拐点），如图 6-1 所示。当系统负载比较空闲时，通过加强系统资源利用率，可以有效地提升吞吐量，并且响应时间受到的影响不大，但是当系统负载到达一定繁忙程度时，再提升吞吐量，响应时间会受到比较大的影响。也就是说，如果超过了这个点，吞吐量提升 1%，可能导致响应时间增加两倍，我们称这个点为拐点，也可以叫平衡点。这个点就是我们要优化的目标，虽然这个点并不是特别容易找到，吞吐量和响应时间的关系也不一定像图 6-1 中描绘得那么陡峭，但是我们在做优化的时候，需要综合考虑，在优化一个指标的时候千万不能忘记其他指标。

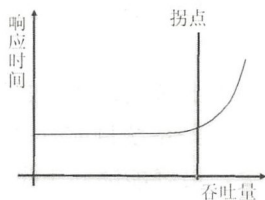


图 6-1 拐点



那么如何找到拐点呢？我们可以通过一组压力测试数据找到。压力测试数据，如表 6-1 所示。我们能够很容易地发现线程数 50 就是拐点。

表 6-1 压力测试数据

线程数	响应时间（ms）	吞吐量（QPS）
1	10.001	134
10	11.321	1568
50	15.982	3893
60	30.834	4034

### 6.4 如何定位瓶颈点

定位性能瓶颈点既是一项非常简单的工作，也是一项非常复杂的工作。当你了解了各个中间件、数据库、网络、CPU、内存等指标后，就能够凭借经验值和几条简单的命令确定出瓶颈点，但是如果你对这些不了解，那将是一项非常复杂的工作。如果要排查性能问题，通常会从以下几个方面入手。

**压力测试。**压力测试主要用于预防性能问题发生，保证性能目标完成。压力测试通常会在测试环境和生产环境进行，但是测试环境测出来的结果往往并不准确。原因主要有两个，一个原因是很难模拟出与生产环境一模一样的数据，当然，有些企业为了解决这个问题，会把生产环境的数据经过脱敏处理导入测试环境，甚至将用户行为日志在测试环境重放。另一个原因是生产环境的基础设施、公共基础服务、依赖服务在压力之下的表现很难被模拟出来。此外，最好是基于场景测试，单独测试某个功能是毫无意义的。目前很多互联网公司都采用线上压测的方式，利用 TCPCopy 等类似工具，在入口复制生产环境的请求进行压测。

**日志分析。**日志分析是当性能问题出现后，定位瓶颈点的方式。如果日志记录比较全面，那么很容易通过日志发现问题。例如，通过错误日志中抛出空指针的异常，可以迅速找到问题点。在微服务架构中，通常调用链日志分析先进行大体定位，然后通过 ELK 收集到的具体业务关联日志、错误日志进行具体分析定位。

**监控工具。**通过工具监控系统实时状态可以快速定位系统性能瓶颈点。当然，如果有统一的监控系统更好，如果没有，也会采用一些命令工具进行定位。

如 Linux 下常用命令行工具。

- dstat 是一个全能系统信息统计工具，是用来替换 vmstat、iostat、netstat、nfsstat 和 ifstat 这些命令的工具。dstat 可以即时刷新、彩色显示，它是一个非常好用的工具。
- sar（system activity reporter）是目前 Linux 上最为全面的系统性能分析工具之一，

可以从多方面对系统的活动进行报告、反馈，包括文件的读写情况、系统调用的使用情况、磁盘 I/O、CPU 效率、内存使用状况、进程活动及 IPC 有关的活动等。

- `netstat` 用于报告 Linux 中网络系统的状态信息，可让你得知整个 Linux 系统的网络情况，如通过 `netstat -an|grep port` 查询某个端口是否被占用。
- `tcpdump` 可以用来查看网络连接的封包内容。它显示了传输过程中封包内容的各种信息，为了使得输出信息更为有用，它允许使用者通过不同的过滤器获取自己想要的信息。
- `lsdf` 是 `list open files` 的缩写，即列出当前系统打开的文件。众所周知，Linux 可以看作一个文件系统，Linux 下任何事物都以文件的形式存在，通过文件不仅仅可以访问常规数据，还可以访问网络连接和硬件。因此通过 `lsdf` 工具能够查看相关列表对系统的监测及排错。

此外，常用的命令还包括 `free`、`iftop`、`Htop`、`vmstat`、`iperf` 等。总之尽量利用工具定位问题。找到瓶颈点之后，对系统进行性能优化的手段有很多，包括同步转异步、阻塞转非阻塞、数据冗余、数据拆分、数据合并、压缩、简化业务环节等。关键取决于应用场景和成本。优化同样需要利用经验，一个经验丰富的程序员可以在短时间内使系统的性能得到数量级的提升。如在数据库中加索引、增加缓存等，效果非常显著。

## 6.5 服务通信优化

在分布式架构中，当服务数量较多时，服务之间的通信对整体性能的影响巨大。

### 6.5.1 同步转异步

一个电商网站的单品页面调用价格、库存、商品等服务综合展示信息，如果是串行调用，则总耗时等于每个服务耗时之和；如果是并行调用，则总耗时等于三个服务耗时的最大值，响应时间差一倍，如图 6-2 所示。因此，采用异步的方式，通常是提升吞吐量非常有效的手段之一。



图 6-2 同步调用与异步调用

同步调用可以理解为串行，异步调用可以理解为并行，当然在最理想状态下可以把异

步看成并行调用。在很多场景中，异步不代表一定可以达到并行效果。

毫无疑问，异步就是启动多个线程，让每个线程去做一部分工作，互不干扰。在前端页面，我们可以利用 ajax 技术实现异步。在后端，一般通过线程池来实现。

异步可以充分利用 CPU 提升吞吐量，降低响应时间，它的代价就是增加了编程、调试的复杂度。

### 6.5.2 阻塞转非阻塞

阻塞调用是指调用结果返回之前，当前线程被挂起，调用线程只有在得到结果之后才会返回，如图 6-3 所示。非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程，如图 6-4 所示。阻塞和非阻塞关注的都是程序在等待调用结果（消息、返回值）时的状态。

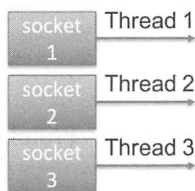


图 6-3 阻塞 IO

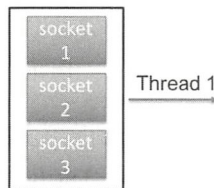


图 6-4 非阻塞 IO

当使用异步调用的时候，如果需要返回结果，有如下三种方式。

- 状态：通过变量实现，需要主线程不断轮询变量结果。
- 通知：通过消息的方式比状态的方式效率高。
- 回调：本质上和通知类似，如 ajax 中的回调函数。

在 Java 中，如果需要返回结果，则可以采用 `futur` 的模式。例如，对用户表进行了水平切分，需要从多个表查询数据时，则可以利用 `futur` 获取结果。因为是异步，调用 `future.get()` 的时候不一定能够获取到结果，如果还没有结果，则会被阻塞。但是在必须获取结果之前，主线程还可以做很多事情，这时候是不受影响的，是非阻塞的状态。

`Future` 模式需要通过轮询或阻塞等待的方式，才能得到结果。这样总是显得不太优雅，更好的方式是通过 `callback`，也就是执行结束的时候异步通知完成状态，然后去 `futur` 中取执行结果。实际上 `futur`、`callback` 这种经典的模型在很多语言里都有了原生的支持，JDK 中虽然有 `futur`，但是并不支持 `callback` 模式。还好 Guava 又给我们打开了一扇窗——`ListenableFuture`。

微服务进行远程通信时，使用非阻塞 I/O 可以解决由于网络时延大、高并发接入等导致的服务端线程数膨胀或者线程被阻塞等问题。



6.5.3 序列化

在图 6-5 中，横轴是线程数，每个线程数从左到右依次列出 Avro、Thrift、Protobuf、gRPC、HTTP+json 的响应时间；纵轴是响应时间，单位为毫秒。我对比了主要的几种 RPC 的响应时间和吞吐量，Avro、Thrift、Protobuf 是一个级别的，相差不大，响应时间在 0.1 毫秒以内，gRPC 的响应时间要比前三者的平均响应时间高出一倍左右，而 HTTP+json 的响应时间接近 1 毫秒。

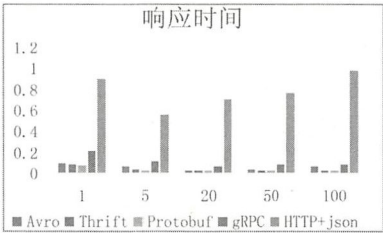


图 6-5 响应时间对比

图 6-6 中，每个线程数从左到右依次列出 Avro、Thrift、Protobuf、gRPC、HTTP+json 的吞吐量。从吞吐量来看，Avro、Thrift、Protobuf 仍然是一个级别的，相差不大，而 gRPC 约为前三者平均吞吐量的三分之一，HTTP+json 约为前三者平均吞吐量的五分之一，如图 6-6 所示。

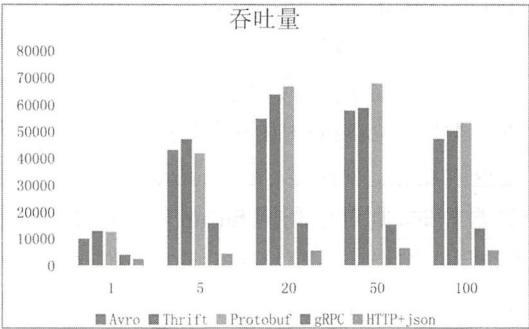


图 6-6 吞吐量对比

当然这和带宽、CPU、数据类型、数据大小等很多因素有关，但根据不同场景测试出的结果可能不一样，但是这不重要，我们不需要如此精确，只要知道大概的数量级就可以在优化阶段大展拳脚了。

从序列化的角度看，序列化后的包大小差距也很大，包越小意味着传输速度越快，带宽占用越小。我简单测试了一下 Protobuf 和 json 序列化后的大小，json 约为 Protobuf 的三

倍左右。Protobuf 压缩优势主要在于对 integer 的压缩，对字符串的压缩比不高，可以序列化之后再行二次压缩，如 GZIP、deflate、LZ4、Snappy 等。更高的压缩比意味着用 CPU 换空间，对 CPU 消耗更大。但是，通常业务服务 CPU 消耗很低，可以利用压缩提升利用率。具体需要根据业务场景而定。

还有其他很多优化点，例如采用长连接，避免重复建立连接导致的性能损失；业务线程和 IO 线程隔离等。

## 6.6 通过消息中间件提升写性能

当规模不断变大以后，数据库通常成为限制系统性能的主要因素。因为如果业务服务的吞吐量不够了，只要扩展业务服务的实例个数就可以了，但是数据库无法很容易地进行伸缩，虽然也可以通过分库分表实现扩展，但是实现成本太高。因此，我们可以换一个思路，通过其他手段降低数据库的压力，那么如何才能平衡数据库的压力呢？

通过消息中间件可以削峰填谷，提升系统整体的吞吐量，如图 6-7 所示。下单操作直接发送到 MQ 即返回，有 MQ 保障消息一定能送达。由于 MQ 的响应时间比关系型数据库短得多，响应时间得以优化。另外，MQ 把依赖关系从强依赖变成若依赖，也就是说，即使订单系统暂时不可用，用户也可以继续下单。

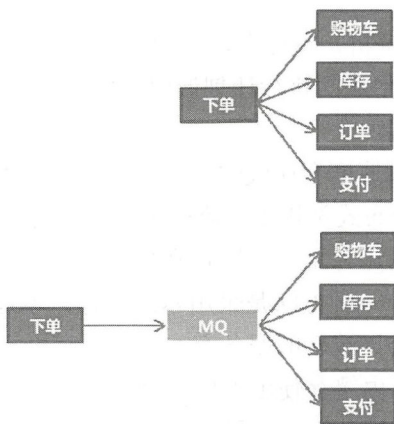


图 6-7 通过消息中间件提升写性能

通过消息中间件可以降低成本，MySQL 5.7 的单表字段个数为 8，数据量为 500 万，写的吞吐量一般在 1 000TPS 左右，而 Kafka 0.9 三节点集群的吞吐量能达到 10 万 TPS 左右，两者的差距大概是两个数量级。当然，MySQL 也可以通过水平分表提升吞吐量，但是水平

分表带来的复杂度非常难解决，消耗的成本也非常高。

当然，使用 MQ 也有一定的问题，需要特别关注一致性的时间窗口，有可能写后读不到，对于有要求的业务来说，是比较致命的，关于一致性可以参考第 7 章的内容。

## 6.7 通过缓存提升读性能

做性能优化时被提到最多的关键词莫过于缓存，特别是在读多写少、高并发的场景中，缓存可以让数据更接近用户，因此，离用户越近的缓存就会越有效。缓存在架构中主要有如下三个作用。

- 减少往返通信（本地缓存）。
- 不需要重新计算缓存结果。
- 平衡数据库的压力。

常见的缓存包括客户端缓存、HTTP 缓存、操作系统缓存、CDN、代理缓存、数据库缓存等。

在数据库中的一次查询大概要经历以下过程。

- (1) 客户端发送一条查询给服务器。
- (2) 服务器检查查询缓存，如果命中缓存，则立即返回存储在缓存中的结果，否则进入下一阶段。
- (3) 服务器端进行 SQL 解析、预处理，再由优化器生成对应的执行计划。
- (4) MySQL 根据优化器生成的执行计划调用存储引擎的 API 来执行查询。
- (5) 将结果返回给客户端。

如果恰好命中缓存，查询速度会比较快。一旦进入复杂的查询，性能跟直接命中缓存的差距会非常大。另外，在分布式架构中，数据库通常是瓶颈点，并且扩展起来非常复杂，利用分布式缓存分担数据库的压力是一个比较常见的做法。

缓存是提升性能的利器。一种方法是采用本地缓存，本地缓存不能共享，会导致比较大的内存浪费，替换节点时会导致新的服务存在大量缓存击穿。在 Java 中，本地缓存分为堆内缓存和堆外缓存两种。堆内缓存使用 JVM 的堆内存存储数据，不需要序列化，速度非常快，缺点也非常明显，频繁的对象创建和销毁会导致大量的 GC，影响业务服务。一般堆内缓存可以直接使用 Map 或 List 实现，也可以用第三方的组件实现，如 Guava Cache、Ehcache 等。为了避免 GC 的问题，可以使用堆外缓存，它不受堆内存的限制，但是需要进行序列化，序列化可以使用 Ehcache 实现。

随着分布式的发展，缓存也开始向分布式缓存发展，这是提升性能的又一种方法。分



布式缓存把业务应用中的缓存抽象到统一的地方进行运维管理，大大简化了业务应用管理本地缓存的复杂度。例如 Redis、Memcached，性能非常高，响应时间为毫秒级，单节点吞吐量约为 10 万 QPS，同等条件下，与数据库几千 QPS 相比是一个非常大的提升。当然，分布式缓存带来了一致性问题，什么时候去更新缓存？如果缓存更新失败、数据库更新成功怎么办？下面详细介绍方法。

### 6.7.1 基于 ConcurrentHashMap 实现本地缓存

在使用其他缓存方式之前，我们需要先了解一下业务场景，采用 ConcurrentHashMap 是否可以满足需求，如果能够满足，则不需要引入更复杂的组件。

基于 ConcurrentHashMap 实现的缓存，首先需要声明一个 ConcurrentHashMap 类，key 通常是由字符串拼接而成，参考如下代码。

```
private static Map<String, Object> cacheMap = new ConcurrentHashMap<String, Object>();
```

以下代码描述的是通过用户 ID 获取用户，如果缓存中有，则直接返回；如果缓存中没有，则从数据库中取出，并写入缓存。

```
public User get(String userID) {
    if (cacheMap.containsKey(userID)) {
        return cacheMap.get(userID);
    }else{
        User user=loadFromDB(userID);
        if (user!=null){
            putUserToZCache(user);
        }
        return user;
    }
}
```

由于 ConcurrentHashMap 是线程安全的，不存在并发问题（注意，同时更新本地缓存和数据库还是存在不一致的可能，只是概率相对较小而已），作为缓存使用是非常合适的，但是这里存在另一个问题，就是缓存数据是在堆内的，需要注意对缓存生命周期的管理，不要让它挤爆了堆内存。为了解决过期时间的问题，需要设置过期时间，例如再定义一个对应的 ConcurrentHashMap 存储过期时间。

```
private static Map<String, Date> expireMap = new ConcurrentHashMap<String, Date>();
```

后台有一个线程不断删除过期的数据，这就变得越来越麻烦，下一节介绍开源的实现方式。

## 6.7.2 基于 Guava Cache 实现本地缓存

Guava Cache 和 ConcurrentHashMap 类似，但也不完全一样。从上一节我们可以知道，ConcurrentHashMap 不会自动清除数据，而 Guava Cache 可以限制内存占用，并且自动回收空间。

例如，你可以用下面的代码构建 LoadingCache。

```
LoadingCache<String, Object> caches = CacheBuilder.newBuilder()
    .maximumSize(1000)
    .concurrencyLevel(5)
    .expireAfterWrite(30, TimeUnit.SECONDS)
    .refreshAfterWrite(1, TimeUnit.MINUTES)
    .build(new CacheLoader<String, Object>() {
        @Override
        public Object load(String key) throws Exception {
            return loadValueByKey(key);
        }
    });
```

LoadingCache 是一个接口，需要通过 CacheBuilder 构建一个 LoadingCache 实现，CacheBuilder 是缓存配置和构建的入口，里面定义了很多配置项。

- maximumSize 定义了缓存的容量大小，这里设置的是 1 000。当缓存数量即将到 1 000 时，则会进行缓存回收，回收最近没有使用或总体上很少使用的缓存项，即 LRU 算法。
- concurrencyLevel 定义了 Segment 的数量，因为 Guava Cache 重写了 ConcurrentHashMap，concurrencyLevel 越大，并发能力越强。
- expireAfterWrite 方法定义了缓存的过期时间，这里是 30 秒，即写入 30 秒之后过期。注意，过期的数据会定期回收。
- refreshAfterWrite 定义了缓存定时刷新时间，这里是每隔 1 分钟缓存值就会被刷新一次。

在 build 方法里，还需要 new 一个自己的 CacheLoader，CacheLoader 需要实现 load、reload、loadAll 方法，用于加载数据。

接下来就可以直接通过 caches.get("key") 获取数据了。

堆外缓存可以借助 EhCache 和 MapDB 框架实现，由于实际使用的比较少，通常采用分布式缓存代替，因此本书不再举例了。在互联网这种大规模、高并发的场景中，分布式缓存才是王道，几乎很少使用本地缓存。分布式缓存可以作为一个服务，由统一的团队去



运维，好于通过组件分散到所有的应用中。

### 6.7.3 缓存的常用模式

下面从应用的角度介绍几种使用缓存的常用模式。

#### Cache-Aside 模式

如何使用缓存呢？下面介绍一下 Cache-Aside 模式，如图 6-8 和 6-9 所示。在 Cache-Aside 模式中，业务服务调用缓存及数据过程如下。

- (1) 判断读取的 key 是否在缓存中。
- (2) 如果命中，则直接返回。
- (3) 如果 key 不在缓存中，则从数据库中将数据读出来。
- (4) 将新的 key-value 写入缓存，然后返回。

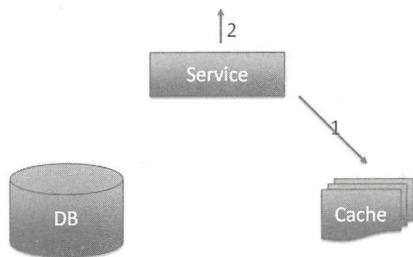


图 6-8 命中缓存

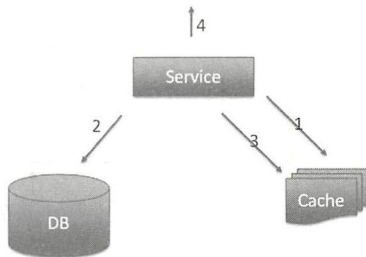


图 6-9 不能命中缓存

缓存失效过程如下。

- (1) 将信息的更新同步到数据库。
- (2) 令缓存中关联的过期数据失效。

如果存在缓存，则证明这里一定是高并发访问的，虽然发生的概率不大，但是毕竟还是有发生概率的，需要格外注意。

写数据的时候，先更新数据库，再更新缓存。如果存在两个并发更新操作，分别先更新数据库，再更新缓存就会发生如图 6-10 所示的问题。p1 将数据库成功更新为 v1 版本，p2 将数据库成功更新为 v2 版本，注意，此时 p1 还没来得及更新缓存。如果发生了 FullGC，p2 先更新了缓存，此时缓存中存在的数是 v2，然后 p1 才更新缓存，此时缓存中存储的数据为 v1。以后的查询操作查到的都是 v1 版本的数据，直到缓存下一次失效。





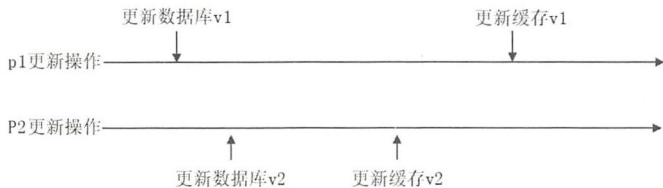


图 6-10 先更新数据库再更新缓存导致问题示例（一）

如果不更新缓存，只删除缓存，则只有读的时候才会取数据放入缓存，就不会发生这个问题了。下面再看一下另外一个问题。

写数据的时候，先删除缓存，再更新数据库。当存在两个并发操作，一个是更新操作 p1，一个是查询操作 p2 时，如果 p1 先删除了缓存中的数据，此时 p2 不能命中缓存，则从数据库中取到 v1 的数据，添加到缓存并返回，此时缓存中存的是 v1 版本的数据，而 p2 把数据库更新为 v2，由于缓存中存在 v1 版的数据，以后的查询操作查到的都是 v1 版本的数据，直到缓存下一次失效，如图 6-11 所示。

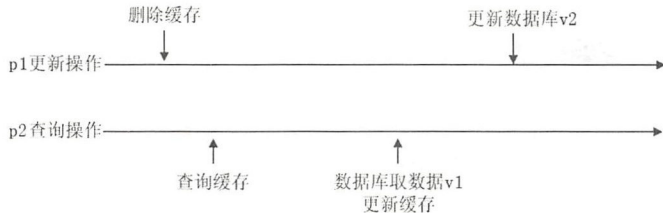


图 6-11 先删除缓存再更新数据库问题示例（二）

既然这样也有问题，我们只能先更新数据库，再删除缓存了。从上面的问题，不难发现这样也存在一定的问题。假设存在两个并发操作，一个是更新操作 p1，一个是查询操作 p2，如果 p2 先查询缓存没有命中，然后从数据库中取数据，得到的版本为 v1，此时发生了 FullGC，p1 更新数据库成功，数据版本变为 v2，然后删除缓存，p1 的所有操作都完成了，p2 醒过来了，开始更新缓存，版本为 v1，由于缓存中存在 v1 版的数据，以后的查询操作查到的都是 v1 版本的数据，直到缓存下一次失效，如图 6-12 所示。

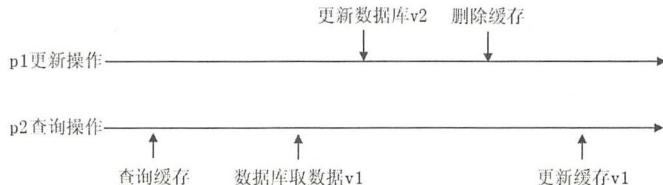


图 6-12 先更新数据库再删除缓存问题示例（三）



这里需要注意的是，这种问题虽然存在理论上的可能，但是发生的概率极低。在分布式环境下，只要是同时更新两个服务，理论上就存在不一致的可能性，包括采用两阶段提交<sup>①</sup>。先更新数据库，再删除缓存，也是我们推荐的方式，可以通过设置过期时间来降低发生不一致的时间长度。

## Cache-As-SoR 模式

从上面的问题，我们能发现，在 Cache-Aside 模式中，应用、缓存和数据库之间逻辑需要由应用自己实现和控制，也就是说业务开发人员需要关注，这其中存在的问题并不简单。而 Cache-As-SoR 模式则是把缓存和数据库看成一个整体，不关心数据如何同步的问题。当然，这只是问题转移了而已，并没有消除。Cache-As-SoR 模式又包括如下三种模式。

- Read Through 模式。上文通过 Guava Cache 实现的代码就属于这种模式，如果命中缓存，则直接返回；如果不能命中，则 Guava Cache 会调用 load 方法加载数据，然后写入缓存。
- Write Through 模式，被称为穿透写模式，写数据的时候，应用直接调用缓存。如果没有命中缓存，则直接更新新数据库，然后返回；如果命中缓存，则更新缓存，由缓存负责数据库持久化。
- Write Behind Caching 模式。和 Linux 下的 Page Cache 的原理一样，它使应用只更新缓存，什么时候更新到数据库，由缓存到数据库的同步策略决定。和 Write Through 模式的区别是，Write Behind Caching 模式是异步的，而 Write Through 模式是同步更新数据库。很明显，Write Behind Caching 模式可以实现批量合并更新，但是问题是可能会丢数据。

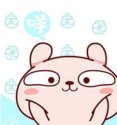
### 6.7.4 应用缓存的常见问题

除了前面提到的不一致的问题，以下几个问题也需要关注一下。

为缓存数据设置合理的过期时间。很多 Cache 实现了过期策略，这些过期策略可以实现数据的更新，使旧数据失效，同时也可以使一定时间没有访问的数据失效。开发人员需要为缓存设置明确的过期时间和过期策略，同时不能让过期时间太短，因为太短的过期时间会让应用频繁地去数据库取数据添加到缓存中，起不到缓解数据库压力的作用。当然，也不能把过期时间设置的太长，太长的过期时间会让缓存存储大量使用频率过低的数据，通过缓存自身的淘汰机制回收空间。

---

<sup>①</sup> 详细内容参见第 7.4.1 节。



为缓存设置回收策略。与数据库相比，绝大多数的缓存容量是很有限的，因此通常情况下，Cache 会移除数据。多数的 Cache 会采用 LRU 的策略来移除缓存中的数据，配置全局的过期属性，可以确保 Cache 消耗的内存资源是高效的。当然，通常不会只配置一个全局的过期策略。常用回收算法包括：FIFO（先进先出）、LRU（最近最少使用）、LFU（最不常用），需要根据业务场景，合理设置。

先预热数据。在新的业务上线的时候，如果不预热数据，所有的请求发送到数据库，当存在大量并发的时候，会导致数据库承受不住压力，大面积宕机。解决方案是在应用启动之前，就将数据库中的数据写入缓存之中，先进行预热。注意，这里缓存的过期时间最好能够错开，否则下一次过期的时候还会存在大面积穿透，解决方案是预热数据的时候在过期时间上加一个随机值。

## 6.8 数据库优化

在 Cloud Native 架构中，通常会把复杂度抽象到公共基础服务中，以此提升架构能力，降低业务开发人员的要求。通过状态外置到缓存、数据库中，来降低业务服务的伸缩复杂度。数据库通常成为各个系统中最难以扩展的点。因此，往往对数据库的优化是非常直接有效的。

以优先级来排序，优化的方式如下。

- 索引、冗余、批量写入。
- 减小锁粒度。
- 减少复杂查询。
- 适当转移事务处理。
- 提升硬件性能。
- 读写分离。
- 分库。
- 垂直分表。
- 水平分表。
- 根据业务情况选择其他数据库，如 NoSQL、时序数据库等。

下面我们以 MySQL 为例，简单描述一下如何提升数据库的性能。

### 6.8.1 通过执行计划分析瓶颈点

在 MySQL 中，可以使用 EXPLAIN 关键字分析 SQL 语句是如何被处理的，可以根据





分析结果定位性能瓶颈。如下示例表示在 MySQL 数据库中对此 select 语句进行分析。

```
EXPLAIN select * from metrics;
```

EXPLAIN 的执行结果如图 6-13 所示。

```
mysql> EXPLAIN select * from metrics;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	<derived2>	NONE	ALL	NONE	NONE	NONE	NONE	171	100.00	NONE
2	DERIVED	global_status	NONE	ALL	NONE	NONE	NONE	NONE	165	100.00	NONE
3	UNION	innodb_metrics	NONE	ALL	NONE	NONE	NONE	NONE	NONE	NONE	Using where
4	UNION	memory_summary_global_by_event_name	NONE	ALL	NONE	NONE	NONE	NONE	320	100.00	NONE
5	SUBQUERY	setup_instruments	NONE	ALL	NONE	NONE	NONE	NONE	1088	11.11	Using where
6	SUBQUERY	setup_instruments	NONE	ALL	NONE	NONE	NONE	NONE	1088	5.56	Using where
7	SUBQUERY	setup_instruments	NONE	ALL	NONE	NONE	NONE	NONE	1088	5.56	Using where
8	UNION	memory_summary_global_by_event_name	NONE	ALL	NONE	NONE	NONE	NONE	320	100.00	NONE
9	SUBQUERY	setup_instruments	NONE	ALL	NONE	NONE	NONE	NONE	1088	11.11	Using where
10	SUBQUERY	setup_instruments	NONE	ALL	NONE	NONE	NONE	NONE	1088	5.56	Using where
11	SUBQUERY	setup_instruments	NONE	ALL	NONE	NONE	NONE	NONE	1088	5.56	Using where
12	UNION	NONE	NONE	ALL	NONE	NONE	NONE	NONE	NONE	NONE	No tables used
13	UNION	NONE	NONE	ALL	NONE	NONE	NONE	NONE	NONE	NONE	No tables used
NONE	UNION RESULT	<union2,3,4,8,12,13>	NONE	ALL	NONE	NONE	NONE	NONE	NONE	NONE	Using temporary; Using filesort

14 rows in set, 1 warning (0.01 sec)

图 6-13 EXPLAIN 的执行结果

对图 6-13 中的字段进行简要说明。

- **select\_type**: 是否为复杂语句。
- **type**: type 为 ALL 表示进行的是全表扫描, 为 index 表示使用了索引。
- **possible\_keys**: 可能可以利用的索引的名字, 如果没有任何可以利用的索引, 则会显示 null, 这个指标对索引优化非常重要。
- **key**: 它显示了 MySQL 实际使用的索引的名字。
- **key\_len**: 索引中被使用部分的长度, 以字节计。
- **Extra**: 查询中每一步实现的额外细节信息, 优化的时候经常被使用。如 Using filesort 表示当 Query 中包含 ORDER BY 操作, 而且无法利用索引完成排序操作的时候, MySQL Query Optimizer 不得不选择相应的排序算法来实现。

## 6.8.2 为搜索字段创建索引

生产环境中的很多故障都和索引有关, 如某电商网站在测试环境测试数据很少, 丝毫测试不出任何问题, 一旦到了生产环境, 大量查询需要进行全表扫描, 导致数据库瞬间崩溃。

索引是数据库存储数据之外维护的一组数据, 索引通常使用 B 树及其变种 B+树实现。当 SQL 语句执行速度较慢, 通过执行计划发现 type 是 ALL, 就可以适当增加索引解决问题。索引为什么能够提升查询效率呢? 用一个生活中的示例来解释, 让你去图书馆寻找一本名叫《持续演进的 Cloud Native》的书, 你可能首先查找计算机类, 然后查找云计算类, 最后查找系统架构类, 通过这个线索缩小检索范围, 这就是典型的按照索引搜索。



索引是一把双刃剑，能够提升读性能，但是会影响写性能，比较适合读多写少的场景。索引具有如下副作用。

- 增、删、改都要维护索引，存在额外的更新成本。
- 索引需要额外存储，消耗空间成本。
- 查询索引也需要额外的时间。

另外，并不是建立了索引就一定能够被查询用到，例如当 where 条件中使用了模糊搜索时，就无法使用索引了。那么哪些列需要建立索引呢？

- 在经常搜索的列上建立索引。
- 在经常用在表之间连接的列上建立索引。
- 在经常需要排序、按照范围搜索的列上建立索引。
- 表的数据量较大时需要索引，数据量越大索引的效果就越好。

哪些字段不需要建立索引呢？

- 对于数值很少的列建立索引的效果不好，如性别，建立索引效果不明显。
- 对于大字段建立索引的效果不好。
- 当对于写性能要求较高时，需要谨慎权衡。

在生产环境添加索引时，MySQL 5.6 以前的版本会导致锁表，所以一定要谨慎。MySQL 5.6 以后的版本支持在线 DDL 操作，在对表进行 alter table 时，对该表的增、删、改、查均不会锁表，但如果在该表被访问时执行 DDL 操作，则会导致锁表，此问题经常出现，需要在操作时查询一下是否有慢查询或者长事务发生阻塞。通过如下命令可以查询数据库的状态。

```
show processlist;
```

执行结果，如图 6-14 所示。如果 Info 列出现 ALTER TABLE `×××` ADD INDEX ×××(×××)语句，就说明发生了阻塞。

```
mysql> show processlist;
+-----+-----+-----+-----+-----+-----+-----+-----+
| Id | User | Host      | db | Command | Time | State | Info |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 50 | root | localhost | sys | Query   | 0    | starting | show processlist |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

图 6-14 执行结果

### 6.8.3 通过慢查询日志分析瓶颈点

在 MySQL 中，慢查询日志主要用来记录响应时间超过阈值的 SQL。一般来说，慢查询发生在单表数据量较大（如一个表的数据量大于几百万），且查询条件的字段没有建立索引时，要匹配查询条件的字段会进行全表扫描，响应时间超过 long\_query\_time（默认为



10 秒) 则为慢查询语句, 会被记录到慢查询日志中, 日志可以被设置为文件或者数据库表。

在命令行输入如下命令可以检查慢查询日志是否被打开。

```
show variables like '%query%';
```

执行结果如图 6-15 所示。如果 `slow_query_log` 的值为 ON, 则开启慢查询日志; 如果 `slow_query_log` 的值为 OFF, 则为关闭慢查询日志。`slow_query_log_file` 的值是记录慢查询日志文件的位置。

Variable_name	Value
binlog_rows_query_log_events	OFF
ft_query_expansion_limit	20
have_query_cache	YES
log_queries_not_using_indexes	OFF
log_throttle_queries_not_using_indexes	0
long_query_time	10.000000
query_alloc_block_size	8192
query_cache_limit	1048576
query_cache_min_res_unit	4096
query_cache_size	1048576
query_cache_type	OFF
query_cache_wlock_invalidate	OFF
query_prealloc_size	8192
slow_query_log	OFF
slow_query_log_file	/usr/local/mysql/data/h1-slow.log

15 rows in set (0.00 sec)

图 6-15 执行结果

## 6.8.4 通过提升硬件能力优化数据库

要优化数据库, 首先应该定位性能瓶颈点。在 Linux 下, 可以使用相关命令查看服务器压力的情况, 详细介绍参见第 6.4 节。另外, 经验值也非常重要, 如 CPU 的延时大概是多少, SSD 和普通磁盘的性能对比。具体 IO 各层次性能对比如表 6-2 所示。

表 6-2 IO 各层次性能对比

	带宽 (吞吐量)	响应时间 (时延)
CPU	20GB/s~400GB/s	0.5 纳秒~15 纳秒
内存	500MB/s~12GB/s	30 纳秒~100 纳秒
SSD 硬盘	50MB/s~2.5GB/s	10 微秒~1 毫秒
SATA 硬盘	50KB/s~600MB/s	5 毫秒~20 毫秒
网卡	10MB/s~10GB/s	100 微秒~1 毫秒

我们发现, SSD 硬盘无论吞吐量还是响应时间都比 SATA 磁盘的高很多。1 个 Master 加 2 个 Slave 结构的 MySQL 使用 SSD 硬盘大概能达到 5 万 QPS, 约为 SATA 磁盘的十倍, 提升硬件能力可以轻易地使系统的吞吐量提升 10 倍。随着 SSD 价格不断下调, 从价格、





开发成本、实际效果来看，适当阶段提升硬件能力绝对是不二之选。目前各大互联网公司的数据库均使用 SSD 硬盘或者 PCIE-FLASH，据说 2012 年的时候微博使用 PCIE-FLASH 支撑了 Feed 系统在春晚时的 3.5 万 QPS。

## 6.9 简化设计

架构是需要全方位考虑的，不可能完全从技术角度去解决问题，不是所有地方都需要高性能，我们还需要考虑架构的复杂度，由此带来的成本，更要权衡代码可读性和可维护性。作为一个技术人员，不能只从技术角度考虑问题，应该思考一下能否将复杂度转移到其他地方，需求是否合理，能否从业务角度解决问题。

### 6.9.1 转移复杂度

思考这样一个场景，假设让我们设计微信红包，无疑节假日时红包的并发量将会非常大，在一张表中存储肯定是不行的，所以一定会拆分到多张表。这样虽然解决了不同红包的并发问题，但是对于一个红包来说，抢红包必定会发生在一张表中，如果通过数据库加锁来实现，那么数据库的压力将会非常大。单纯从优化数据库的角度去实现，问题会越来越复杂。如果把这些抢红包的请求进行排序，串行修改数据库，那么数据库就不需要加锁了，数据库的压力也会小很多。我们可以把关于同一个红包的请求都路由到同一个服务中进行处理，也就是根据红包 ID 进行哈希，然后在一个服务中进行排序处理即可，这种做法缓解了数据库的压力。

另外，正常的逻辑肯定是在拆红包的时候才去计算每个红包的大小，随机出一个数字。仔细分析一下业务特点不难发现，红包拆分的量远远大于红包发送的量，拆红包的压力远大于发红包的压力。我们完全可以在发红包的时候就根据拆分的份数随机给出具体的数额，抢红包的时候直接返回一份，降低了抢红包的压力，而且这样做并没有降低用户体验。

### 6.9.2 从业务角度优化

我们仍然通过一个例子来说明这个问题，如 12306 网站刚上线的时候，大家常常认为在 12306 网站买火车票像秒杀，实际上比秒杀复杂得多。在秒杀场景中，无论前端放过来多少流量，后端都可以根据库存去抛弃多余的量，如秒杀 10 部手机，只需要放过来 1 000 个请求，我们认为这 1 000 个请求足以完成秒杀，其余的流量直接返回，秒杀结束。但是 12306 网站并非如此，首先要在固定时间放出所有的票，其次用户买票的起始站点都不一样，能得出很多种排列组合的可能。



当然，大家也看到了 12306 网站是怎么优化的。首先，分时段放票，不必一次放出所有的票，这样可以把一次秒杀分成很多次，降低某一个时间点的压力，每个省的放票时间错开，这样也可以减少系统的并发压力。其次，票的具体剩余数量可以用“有票、无票、少量”等字眼代替，虽然不如显示具体数量体验好，但是能满足基本要求。最后，采用排队的方式，满足最终一致性即可，查询的时候有票，不一定能买到，提交可以进入队列进行排队。这里我认为还可以采用预售的方式，收集整体的需求，或者通过各种数据进行预测，然后进行全国调度，效果会更好。

再举一个例子，如果你平时不看微信群消息，那么打开群的时候是否要把所有的消息都推送过来呢？如果一次性都推送过来压力会很大，那么可以只推送一部分，当用户有意愿继续阅读的时候，如向上滑动屏幕的时候，再去服务器取更多的消息。

以上都是从业务角度实现的简化设计。



# 7

## 第 7 章 一致性设计

一致性问题在大多数场景中发生的概率很低，往往被忽略。为什么我们在大部分场景中要放弃一致性呢？放弃一致性是不是就一定会出问题呢？需要一致的地方如何解决？带着这些问题，让我们由浅入深地阅读本章内容。

### 7.1 问题起源

一致性问题不只存在于计算机领域，生活中也充满了一致性问题，图 7-1 中的“两军问题”就是一个例子。

假设现在“红军”有两支部队正在攻击“蓝军”，“蓝军”位于一个山谷中，完全将两支“红军”分开。如果要取得胜利，那么两支“红军”必须同时发起攻击，“红军”唯一的通信方式就是穿过“蓝军”送信。如果“红军”的传令兵被“蓝军”截获，则有可能导致“红军”失败。此时如果“红军 1”拟定了作战计划，派传令兵送到“红军 2”，“红军 2”收到后必须再派传令兵告诉“红军 1”已经同意，否则“红军 1”不知道“红军 2”是否同意了。而“红军 2”派通信员回复“红军 1”的消息也有可能被截获，所以“红军 1”还要派传令兵回复“红军 2”，总之最后一次是否被截获是不知道的。这是一个无解的问题。



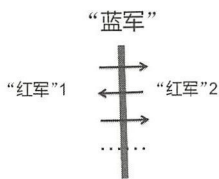


图 7-1 两军问题

两军问题及其无解性证明在 1975 年由 E. A. Akkoyunlu、K. Ekanadham 和 R. V. Huber 首次提出，写进《网络通信设计的约束与权衡》一文中。两军问题是计算机领域的一个思想实验，用来阐述在一个不可靠的通信链路上试图通过通信以达成一致，这是存在缺陷和困难的<sup>①</sup>。

另外一个更著名的问题是拜占庭将军问题，此问题经常出现在计算机网络课程里。拜占庭将军问题很早就被提出，但是由于太复杂，并没有被普及，后来为了普及，采用故事的方式来说明问题，并命名为“拜占庭将军问题”<sup>②</sup>。

这个问题是这样描述的，假设现在有 11 支军队攻打一个城堡，他们可以根据当时的情况判断是进攻还是撤退。如果各支军队行动不统一可能会导致失败，那么怎么保证他们一起进攻，或一起撤退呢？很显然，他们可以通过传令兵进行通信，最简单的方式是让所有人投票，通过少数服从多数的方式确定最终的方案。但是这 11 支部队中可能存在间谍，如果有 5 位判断进攻，4 位判断撤退，假设两位间谍也投撤退票，最终做出错误的决定——全部撤退，但是结果是 11 支军队的行为是一致的，这完全可以接受。

也许在单体架构中，一致性问题并不是特别严重。但是在微服务架构中，服务数量骤增，跨服务调用更是非常普遍，为了实现关注点分离，每个服务独享一个数据库，结果导致出现不一致的概率大增，当然，这是相对单体架构而言。

## 7.2 基础理论

要说明一致性的设计思想，就不得不先说明相关的理论基础。由于分布式的基础理论博大精深，无法进行深入的介绍，本文只能从通俗易懂的角度，挑选几个常用的理论进行简单介绍。

① 来自《百度百科》。

② Lamport 于 1980 年发表的 *Reaching agreement in the presence of faults* 中提出了第一个解决拜占庭将军问题的算法。



### 7.2.1 什么是分布式事务

要理解分布式事务，得先从单机事务开始。事务的本质就是本地任务如何把本地数据共享给其他任务？如何处理并发和锁？即使在单机的情况下，多个线程同时去读数据、写数据，同样也会出现不一致的问题。

举个例子，对于数据库，以下操作就是一个事务单元。

- 读一条数据。
- 删除一张表。
- 更新一个字段和相对应的索引。
- 同时更新两张表。
- .....

要理解事务的概念就必须了解一下事务的四大特征。下面通过一个银行转账的例子来说明事务的特征。

事务的四大特征如下所示。

(1) 原子性 (Atomicity) 是指通过事务保证所有操作是一个不可分割的单元，要么全部成功，要么全部失败。

举个例子：假设 A (10 元) 要向 B (5 元) 转账 5 元，实际发生的步骤如下。

Step1: 查询 A 账户是否大于 5 元，如果大于 5 元进行下一步，状态记为 (A10, B5)。

Step2: A 账户减 5 元，状态记为 (A5, B5)。

Step3: 给 B 账户加 5 元，状态记为 (A5, B10)。

任何一个步骤发生了错误，都会导致整个事务回滚，数据库会通过日志记录回滚操作的数据。

(2) 一致性 (Consistency) 是指通过事务保证数据从一种状态变化到另一种状态。至少在事务结束前，所有数据都处于有效状态。

上例中的关键问题是，如果进行到 Step2，另外一个线程来读取数据，可能会读到 A 账户 5 元，B 账户 5 元，这时就产生了不一致。如果要保证强一致性，避免这个情况的发生，就需要在 Step1 之前加锁，在 Step3 之后解锁，保证中间不会有其他线程读写数据。假设在 Step2 出现读请求该如何处理？有两种方式，一是等到事务结束；二是直接读取事务开始之前的数据。一致性的核心是事务处理的中间状态不可见，要做到这一点就必须加锁，一旦加锁，整个系统的性能将受到极大挑战。

(3) 隔离性 (Isolation) 是指事务内的操作不受其他操作影响，当多个事务同时处理同一个数据的时候，多个事务之间是互不影响的。

前面提到，如果按照强一致性的要求会导致系统性能大幅度降低，而通过调整隔离性



可以缓解这个问题。

为了方便说明，我们根据 Lamport 定义的两个事件之间发生的关系，即 happen-before 关系来进行说明。事务之间的关系可以通过 happen-before 表达为如下四种。

- 读写。
- 写读。
- 读读。
- 写写。

ANSI SQL92 标准定义的四种事务隔离级别如下所示。

- 未提交读（Read uncommitted）。

未提交读指一个事务可以读到另一个未提交事务的数据，隔离级别比较低。这种情况下，读读、读写、写读是可以并行的，唯有写写不可并行。

未提交读会导致脏读。例如原子性中提到的，可以读到中间状态，“Step2: A 账户减 5 元，状态记为 (A5, B5)”。

- 提交读（Read committed）。

提交读指一个事务不能读取另外一个未提交事务的数据。在这种情况下，读读、读写可以并行。

提交读保证了读到的任何数据都是已经提交的数据，可以避免脏读，但是不能保证事务重新读的时候能读到相同的数据。因为在每次数据读完之后，其他事务可以修改刚才读到的数据，所以提交读不能解决不可重复读的问题。

- 可重复读（Repeatable reads）。

为了解决不可重复读的问题，隔离级别再上升一下，变成可重复读。也就是说，事务 1 在读取某条数据的时候，事务 2 也可以读取，因为数据无变化；事务 1 在读取某条数据的时候，事务 2 不能修改该数据；事务 1 在修改某条数据的时候，事务 2 不能修改该记录。这种情况下，只有读读可以并行。

因为如果事务 1 读取一个列表，例如 `select * from A where x<10`，如果返回 10 条记录，事务 1 会对这 10 条记录加行级共享锁，此时另外一个事务 2 执行 `insert into A values (...9...)`。因为只是行级共享锁，并不是表级锁，事务 1 在第二次读取的时候返回 11 条记录（注意，事务 1 的两次读取是在一个事务内，此处容易混淆）。这个问题被称为幻读。因此，可重复读不能解决幻读问题。

- 可序列化（Serializable）。

可序列化是最高的隔离级别。可序列化可以解决幻读的问题。在这种情况下，所有的操作都必须串行。





事务 1 在读取数据时增加表级锁，事务结束才会释放。此时，事务 2 可以读数据，但是不能插入、更新、删除数据。

事务 1 在更新数据时增加表级排他锁，事务结束才会释放。此时，事务 2 不能读写。

可序列化虽然级别更高，解决了脏读、不可重复读、幻读等问题，但是也会带来性能低的问题，性能低在某种程度上也可以理解为可用性低。

为了解决上述问题，目前主流数据库普遍在“92 标准”上扩展了隔离级别，就是大名鼎鼎的多版本并发控制（MVCC），它的核心思路就是 Copy On Write。

（4）持久性（Durability）是指事务被提交后，应该持久化，永久保存下来。

要实现上述四个特征（被简称为 ACID）并不是一件容易的事情。经验表明，在保证 ACID 的情况下，往往可用性和性能比较差，所以一般分布式系统会尽量放宽隔离性的要求。

分布式事务就是在分布式场景下，如何实现单机事务的 ACID 特性。

分布式思想给系统带来了扩展能力和可用性，同时也给编程带来更多的挑战，因为网络是不确定的。虽然知道每个节点的成功和失败，但是无法保证所有的节点要么全部成功，要么全部失败。

## 7.2.2 CAP 定理

在计算机科学中，CAP 定理又被称作布鲁尔定理（Brewer's theorem）。它认为对于一个分布式计算系统来说，不可能同时满足以下三点。

- 一致性（Consistence）。
- 可用性（Availability）。
- 分区容错性（Partition tolerance）。

在大型分布式系统实践中，分布式意味着必须满足分区容错性，也就是 P。为了追求更高的可用性，在一致性上会做一定的妥协，通常会选择 AP。CAP 定理如图 7-2 所示。

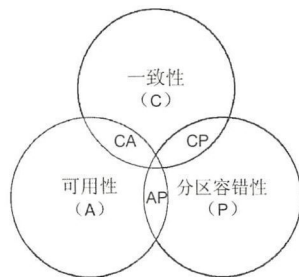


图 7-2 CAP 定理

### 7.2.3 BASE 理论

基于对大规模分布式系统的实践总结，eBay 的架构师 Dan Pritchett 在 ACM（国际计算机学会）上发表文章提出了 BASE 理论，BASE 理论是对 CAP 定理的延伸。

- BA: Basically Available, 基本可用。
- S: Soft state, 软状态。
- E: Eventually consistent, 最终一致。

BASE 理论的核心思想是：如果无法做到强一致性，或者做到强一致性要付出很大的代价，那么应用可以根据自身业务特点，采用适当的方式来使系统达到最终一致性，只要对最终用户没有影响，或者影响是可接受的即可。

### 7.2.4 Quorum 机制（NWR 模型）

从前面的章节我们得知，在分布式场景中，扩展性、可靠性可以通过复制数据副本实现。如果多个服务分别向三个节点写数据，为了保证强一致，就必须要求三个节点全部写成功才返回。这样在读的时候可以读任意节点，就不会有不一致的情况了。但是，同步写三个节点的性能较低，如果换一个思路，一致性并不一定要在写数据的时候完成，可以在读的阶段决策，只要每次能读到最新的版本就可以了。这就是 Quorum 机制的核心思想。

简单来说，Quorum 机制就是要满足公式  $W+R>N$ ，式中  $N$  代表备份个数， $W$  代表要写入至少  $W$  份才认为成功， $R$  表示至少读取  $R$  个备份。这个公式把选择权交给了业务用户，让用户来做出最终决策。

NWR 原理，如图 7-3 所示。两个进程同时往三个节点写数据，v1 表示版本 1，v2 表示版本 2，如何保证每次读取都至少读到一个最新的版本呢？

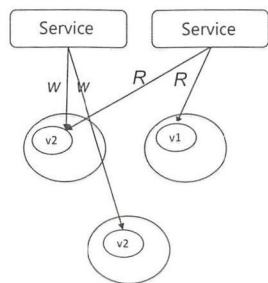


图 7-3 NWR 原理

假设  $N=3$ ,  $W=1$ ,  $R=1$ ,  $W+R<N$ , 在节点 1 写入，节点 2 读取，无法读到最新的数据。

假设  $N=3$ ,  $W=1$ ,  $R=2$ ,  $W+R=N$ , 在节点 1 写入，节点 2、3 读取，无法读到最新的数据。

假设  $N=3$ ,  $W=2$ ,  $R=2$ ,  $W+R>N$ , 写入任意两个节点，读取任意两个节点，一定会读取到

最新版本的数据。

假设  $N=3$ ,  $W=3$ ,  $R=1$ ,  $W+R > N$ , 同时写入所有节点, 则读取任意节点就可以得到最新的数据。

当  $R=1$  且  $W=N$  时, 适合读多写少的场景, 读操作是最优的。当  $W=1$  且  $R=N$ , 适合写多读少的场景, 可以得到非常快的写操作。可以根据读、写比例和应用场景灵活设置  $W$ 、 $R$ 。

但是, 这里面还存在一个写入冲突的问题, 举例说明一下: 假设  $N=3$ ,  $W=1$ , 一共有三个节点, 只要写入一个就认为成功。如果第一次写入 A 节点, 对变量 a 进行减 1 操作, 变量 a 在 A 节点上由 10 变成了 9, 记录版本为 v2, 变量 a 还没来得及同步到另外两个节点, a 在 B、C 节点上的值还是 10; 第二次写入 B 节点, 同样对 B 节点进行减 1 操作, 变量 a 在 B 上的值变为 9, 版本为 v3, 根据  $W+R > N$  规则, 要读取三个节点, 同时得到了 v2、v3 版本的数据, 这时候就需要合并数据, 处理冲突。由于 v3 版本更新, 就会覆盖 v2 版本, 结果  $a=9$ , 但是实际执行了两次减 1 操作。

如果要解决这个问题, 就要同时满足公式  $W > N/2$ 。这样能保证每次写入都能和上次写入有交集, 变成了一个乐观锁, 只有超过半数写成功才算成功。

Aurora 是 AWS 的分布式关系型数据库, 它的存储层就是基于 Quorum 协议的, 除满足  $W+R > N$  外, 还有一个要求就是  $W > N/2$ 。Aurora 会部署在三个 AZ (AvailabilityZone) 上, 每个 AZ 包含了 2 个副本, 总共 6 个副本, 至少 4 个实例写成功才算成功, 至少读三份数据。这样即使任意一个 AZ 不可用, 还有 4 个副本, 不会丢失数据, 不影响读写。此外, 当任意一个 AZ 不可用的同时, 另外两个 AZ 中的一个副本节点也不可用了, 这样只会影响写, 不会影响读。由于在同一个机房通过备份恢复一个副本节点会很快, Aurora 采用万兆网卡, 可以在 10 秒内恢复一个 10GB 的副本。

### 7.2.5 租约机制 (Lease)

租约机制, 如图 7-4 所示。如果现在我们有三个节点, 为了实现一致性, 要确保有且只有一个是 Leader, 另外两个为 Follower, 只有 Leader 是可写的, Follower 只能读。管理节点 M 通过心跳判断各个节点的状态, 用 M 去指定 Leader, 一旦 Leader 死掉, 就可以重新指定一个 Leader。

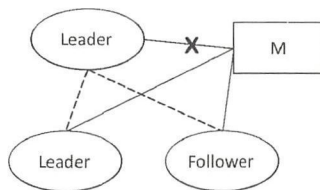


图 7-4 租约机制





我们再来看另外一个心跳机制不容易解决的问题。假设 Leader 不可用，剩下两个 Follower，此时 M 又在另外两个 Follower 里选出了一个 Leader，而原 Leader 并不一定真的不可用了，也许是因为网络故障导致 M 和 Leader 之间的网断了，但是 Leader 和另外两个 Follower 的网并没有断开。这时候 M 在另外两个 Follower 中重新指定了一个 Leader，就会出现“双 Leader”，两个 Leader 都可以写，结果产生不一致，俗称“脑裂问题”。

对于上面两种情况，有两种解决方案，一种是采用投票机制（Paxos 算法）；另外一种则是采用租约机制——Lease。

租约机制的核心就是在一定时间内将权力下放。M 可以给 Leader 的写权利设定一个有效期，在有效期内，M 不去重新选主，就算 Leader 已经不正常，我们也认为它是正常的。这样即使出了问题，最多就是有效期时间段内的不可用的问题，避免了脑裂问题。

另外一个问题，在分布式存储系统中，通常在 Leader 中存放元数据，此时，Leader 的压力非常大。为了缓解这个问题，我们可以给 Follower 同步元数据，并给出一个有效期，保证在有效期内元数据不发生变化，以此减轻 Leader 的压力。Google 的论文中提到的 GFS（Google File System），包含 Master 和 Chunkserver，就是通过租约机制来减轻 Master 压力的。

租约机制能够容忍网络故障，一旦租约被颁发，客户端就可以按照租约去执行，有效期之内都不需要依赖网络，但是租约机制也存在丢数据的风险。假设有效期内 Leader 和 Follower、M 发生了断网，但是在租约之内，业务服务很可能会继续写数据到 Leader 中，租约过后，又有了新的 Leader，老 Leader 上的数据并未得到及时同步，此时很难合并数据，除非通过人工干预，但是通常为了可用性会选择放弃老的 Leader 之上的数据，可以参考 Etcd。

## 7.2.6 状态机（Replicated State Machine）

在分布式环境下，如果要保证更高的可用性，更好的容错，就必须建立多个副本。举个例子，为了对抗“挖掘机”（指黑客软件），就必须在多个数据中心同步数据，如何保证在跨数据中心的情况下，数据库的数据一致？一个简单的办法就是执行相同的命令，也就是说在每个数据中心都有相同的初始状态，执行相同的命令，并且命令的顺序保持一致，这样就可以保证最终数据也相同了。

总之，在任意时刻，要想让所有节点达成一致，有两种方式。

- 所有节点数据都是一致的，客户端可以访问任意节点。
- 客户端只访问主节点，主节点有最终决策权。

首先，如何保证所有命令按顺序执行？

这一点比较容易满足，只能通过 Leader 写数据，而 Leader 节点只有一个，只要通过选主算法选出 Leader，就可以保证所有的命令都按照顺序执行。如果写请求被分发到了



Follower，则将写请求转发到 Leader，实际上就是 Leader 顺序写 log，这里写的方式是 append-only 的，也就是说只能在末尾追加，不能修改，也不能删除。Follower 定时拉取 log，或者是写 Leader 的时候，同时写入 Follower 成功再返回，这个需要根据业务场景来确定，可以参考 MySQL 主从同步、半同步复制、Kafka 主从同步等。

其次，发生故障该如何切换？

现在已经有好几种选择了，例如 Paxos、Raft。Raft 相对来说更简单一些，分为两种情况，Follower 发生故障时，不受任何影响；而如果 Leader 发生故障，则需要重新选 Leader，这个时候所有节点都变为“待选主”状态。需要注意的是，一旦选主成功，而老的主又恢复过来了，这时候老的主是不能随便加 Leader 进来的，因为老的主必须和新的主保持一致，而随便加 Leader 会造成不一致。

还有一个问题是，如果一直写 log 来同步状态，那么 log 会越来越大，一旦重做 log 会导致大量计算，而且比较浪费存储空间，比较好的办法是结合 SnapShot（快照），也就是一定时间后把所有结果数据存为一个快照，把快照作为初始状态继续写 log。一旦需要重做 log，会先加载快照，然后重做 log。Redis Cluster 最新的改进就采用了这种方法。

为了解决此类问题，Replicated State Machine 应运而生。这个理论是著名图灵奖得主 Lamport 在 1978 年的论文中提出来的，现在好多分布式系统的理论都是基于这篇论文提出的。

Replicated State Machine 的一个典型应用案例就是 MySQL 同步，因为 MySQL 本身只支持 Master-Slave 结构。假设我们想同时写入三个节点，我们就可以在写入 MySQL 之前先通过 Proxy 写 log，三个节点的 Proxy 之间通过 Raft 选出 Leader，Leader 接受写请求，然后通过 log 实现各个 replica（副本）的同步。这样可以保证读取到最新的数据。

## 7.3 分布式系统的一致性分类

我们可以从两个方面着手来提升系统的可用性和可扩展性。

- 建立多个副本。可以把副本放到不同的物理机、机架、机房、地域，当一个副本失效时，可以让请求转到其他副本。
- 对数据进行分区。复制多个副本解决了读的性能问题，但是无法解决写的性能问题。

根据关键字进行分片，实现数据分布式，进一步提升系统的写性能。

以上两种方式最复杂的问题就是如何保证一致性。一致性问题一直以来都是分布式系统的痛点，因为本身一致性的场景有很多，并不是所有的系统都要求是强一致的，强一致需要极大的成本，要根据系统的容忍度适当放宽一致性的要求。

在很多人看来，银行间转账应该是强一致的，但是仔细分析会发现，小王给小张转账

1 千元，小王的账户扣除了 1 千元，此时小张的账户并不一定会收到 1 千元，可能会存在一个不一致的时间窗口。也就是小王的账户扣除了 1 千元，而小张的账户还没收到 1 千元。另外一个例子，在 12306 网站买火车票的功能也未必是强一致的，如果你在 12306 上发现去某地的火车票还剩 10 张，发起订一张火车票的请求，系统给你返回的可能是“正在排队，剩余 10 张票，现在有 15 个人在购买”，可能需要你去查询未完成的订单，并不给你及时返回成功或失败的结果。如果有人退了一张票，系统也没有把退票立即返回到票池中。这里明显也存在不一致的时间窗口。

学术界的一致性模型主要从以下两个角度去分类<sup>①</sup>。

- 以数据为中心的一致性模型。它从全局考虑，如发了一条新闻，所有人都能看见。
- 以客户为中心的一致性模型。从用户的角度考虑，如甲购买了一个商品，乙也购买了一个商品，两个买家之间没有任何关系，不用保持一致，只要与甲或乙有关系的数据保持一致就行了。

### 7.3.1 以数据为中心的一致性模型

以数据为中心的一致性是从数据存储的角度出发的，包括数据库、文件等。

实际上，一致性是进程和数据之间的规则约定。在图 7-5 中，假设有 3 个进程，用 p1、p2、p3 来表示，跟进程对应的有多份存储，用 a 来表示变量，W(a)1 来表示进程写入 a=1，R(a)1 表示读取 a 的值，结果为 1。

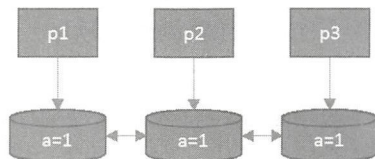


图 7-5 一致性举例

以下一致性模型是异步执行的，没有任何同步操作。

#### 1. 严格一致性 (Strict Consistency)

严格一致性要求任何写操作都能立刻同步到其他所有进程中，任何读操作都能读取到最新的修改。要实现这一点，要存在一个全局时钟，但是在分布式场景下很难做到，所以目前严格一致性在实际生产环境中还无法实现。

严格一致性，如图 7-6 所示。所有的对变量 a 的读取都能够读取到最新值，无论是否

<sup>①</sup> 参考了 TODO。



在一个进程上，如果从客户端发起两个请求，先发起的必须先到达，后发起的必须后到达。举个例子说明一下，假设 A、B 两个人同时去两个不同的邮局寄信，并且寄到同一个人手中，如果 A 先完成了邮寄，无论中间发生任何变化，那么都必须保证 A 的信先被收到。如果没有全局时钟，那么你就无法知道是 A 先完成的还是 B 先完成的。

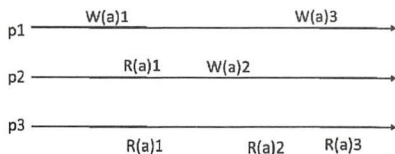


图 7-6 严格一致性

严格一致性很难实现，原因是全局时钟难以实现。注意，这里所说的严格一致性不等于强一致性。

## 2. 顺序一致性（Sequential Consistency）

全局时钟导致严格一致性很难实现，顺序一致性放弃了全局时钟的约束，改为分布式逻辑时钟实现。顺序一致性是指所有的进程以相同的顺序看到所有的修改。读操作未必能及时得到此前其他进程对同一数据的写更新，但是每个进程读到的该数据的不同值的顺序是一致的。举例说明一下，你在手机上看到的聊天顺序和在电脑上看到的聊天顺序是否一致，如果一致，则满足了顺序一致性，通常在消息存储的时候会为所有的消息生成一个唯一的 ID，消息展示是按照 ID 排序的，这样就满足了顺序一致性。

在图 7-7 中，变量 a 的写操作发生在三个不同的进程中，如果按照严格一致性，变量 a 的结果应该是 3。在顺序一致性的场景下，有一个确定顺序的过程，由于没有全局时钟，后执行的进程未必最后到达，可能导致最后 a 的值是 1。只要三个进程读取到的顺序是一致的，哪怕最后一次读取到的是 1，也是满足顺序一致性的。但是在图 7-8 中，非常明显，三个进程读取出来的顺序是不一致的，p2 读取到的结果顺序和 p1、p3 的不同，不能满足顺序一致性。

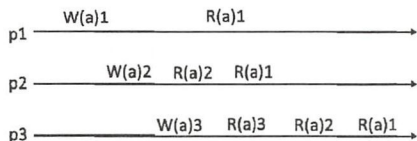


图 7-7 满足顺序一致性

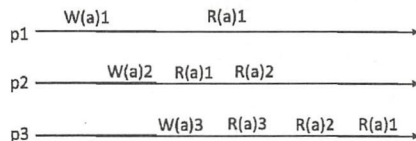


图 7-8 不满足顺序一致性

顺序一致性需要额外实现一个逻辑时钟服务，有额外的性能开销。

### 3. 因果一致性 (Causal Consistency)

因果一致性是一种弱化的顺序一致性。所有进程必须以相同的顺序看到具有潜在因果关系的写操作，不同进程可以以不同的顺序看到并发的写操作。

p2 中的读写存在因果关系， $W(a)=2$  可能是先读到了  $a=1$  计算后的结果，如图 7-9 所示。所以 p3 中读到的顺序不满足要求，在读到  $a=2$  之后，不能再读到  $a=1$ 。因为图 7-10 中不存在因果关系，所以并不做要求，满足因果一致性。举例说明一下，如群里有人问你“中午吃饭了吗（写数据）？”你看到了这句话（发生了因果关系），回答“吃了（写数据）”，另一个人不可能先看到“吃了”，然后才看到“中午吃饭了吗”的提问。

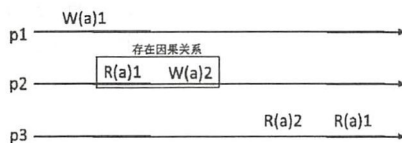


图 7-9 不满足因果一致性

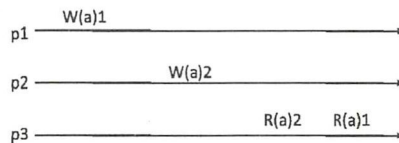


图 7-10 满足因果一致性

相比于顺序一致性，因果一致性只强调存在因果关系的操作被看到的顺序一致，没有因果关系的操作被看到的顺序可以不一致。因果一致性并不对并发操作排序，在分布式系统中，如果  $a$  和  $b$  是并发的，且  $a$  和  $b$  是两个不相关的操作，没有任何因果关系，那么它们在分布式系统中就不必遵循任何顺序了，这样就避免了在它们之间使用因果这种串行化方式。因为串行化会成为系统的性能瓶颈，因果一致性提升了并行的概率。

### 4. FIFO 一致性 (FIFO Consistency)

FIFO 一致性是在因果一致性模型上的进一步弱化。FIFO 一致性要求所有进程以某个单一进程提出写操作的顺序看到这些写操作，但是不同进程可以以不同的顺序看到不同的进程提出的写操作。通俗来讲，就是要求在一个进程内，所有的写操作必须在对外可以被看到的时候是一致的，但是两个进程的写操作的顺序被看到的时候可以不同，就算是有因果关系也不保证。

p3 在任何时刻读取到的  $a$  的值的顺序，只要保证在 p1、p2 各自的写顺序就可以了。也就是要保证 3 在 1 的后面，4 在 2 的后面，哪怕 4 和 3 存在因果关系，如图 7-11 所示。图 7-12 的两种情况违背了单进程的写顺序。

在实际过程中，FIFO 一致性可以适当变化，根据业务情况把有因果关系的数据放到一个分区，在同一个进程写入。例如某电商价格系统在设置价格时会写入 Kafka，可以根据商品 ID 分区，在消费阶段，一个分区最多只有一个消费者。在同一个分区 Kafka 是保证顺序消费的，但是在不同分区没有任何约束。在满足业务一致性的前提下，极大地提升了并发处理能力。

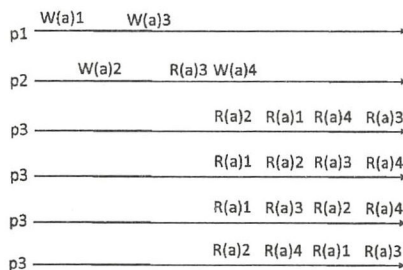


图 7-11 满足 FIFO 一致性

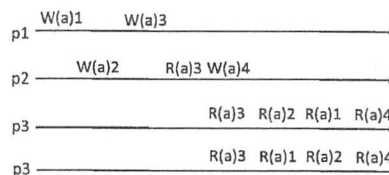


图 7-12 不满足 FIFO 一致性

以下三个一致性是需要同步变量的。也就是说，当一个进程对变量设置值的时候，不保证其他进程什么时候能够看到值的变化，但是当执行了一次同步操作后，所有进程会看到最新的值。此处不再详细介绍。

- 弱一致性（Weak Consistency）。
- 释放一致性（Release Consistency）。
- 入口一致性（Entry Consistency）。

### 7.3.2 以用户为中心的一致性模型

在实际业务要求中，很多时候并不要求系统内所有的数据都保持一致，例如在线的日记本，业务只要求基于这一个用户满足一致性即可；不需要关心整体。这就是所谓的以用户为中心的一致性。

以下一致性模型适应的场景为不会同时发生更新操作，或者同时发生更新操作时能够比较容易地化解。因为这里的数据更新默认有一个与之关联的所有者，此所有者拥有唯一被允许修改数据的权限，可以按照用户 ID 进行路由。通过这种方式，可以大概率避免写-写冲突，除非出现重新负载均衡的过程。

我们简化一下场景，假设有一个进程对一份数据的一个副本进行操作，更新要传递给其他副本，如图 7-13 所示。注意，这里是单个进程，而前面以数据为中心的一致性模型中的例子是多个进程。

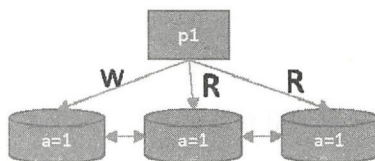


图 7-13 一致性举例



### 1. 单调读一致性 (Monotonic-read Consistency)

单调读一致性是指如果一个进程读取数据项  $a$  的值，那么该进程对  $a$  执行的任何后续读操作总是得到第一次读取的那个值或更新的值。

在图 7-14 中，如果变量  $a$  的初始值为 0，一个进程  $P$  在  $L1$  副本上做加 1 操作，得到结果  $a=1$ ，此时  $a=1$  从  $L1$  副本传递到  $L2$ ，进程  $P$  又在  $L2$  副本做加 1 操作，但是  $L2$  副本中的  $a$  并没有立即传递到  $L1$ ，进程  $P$  在  $L1$  上读到  $a$  仍然是 1，然后进程  $P$  又在  $L2$  上读到  $a=2$ ，明显比上一次读到的值更新，符合单调读一致性。而图 7-15 中进程  $P$  先在  $L1$  加 1，随后读取  $a=1$ ，此时变量  $a$  的值还没有传递到  $L2$ ，进程  $P$  在  $L2$  读到的  $a=0$ ，读到了更老的版本，显然不符合单调读一致性。

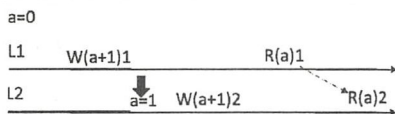


图 7-14 满足单调读一致性

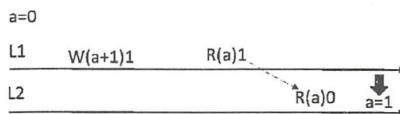


图 7-15 不满足单调读一致性

单调读一致性强调任何时刻不能读到比以前读到的数据还旧的数据。在实际业务中，某一用户读到了一个审批流程，在页面看到已经进行到了第四步，刷新了一下，可能路由到了另外一个数据副本，又回到了第三步，这个流程便产生了错误。

### 2. 单调写一致性 (Monotonic-write Consistency)

单调写一致性是指一个进程对数据项  $a$  执行的写操作必须在该进程对  $a$  执行任何后续写操作前完成。

如果变量  $a$  的初始值为 0，那么一个进程  $P$  在  $L1$  副本上做加 1 操作，得到结果  $a=1$ 。在  $a=1$  从  $L1$  副本传递到  $L2$  副本之后，进程  $P$  又对  $a$  做加 1 操作，此时  $a=2$ ，满足单调写一致性的要求，如图 7-16 所示。 $a$  加 1 操作并没有及时传递，导致进程  $P$  在  $L2$  做加 1 操作时，得到的结果有误，如图 7-17 所示。

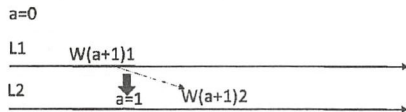


图 7-16 满足单调写一致性

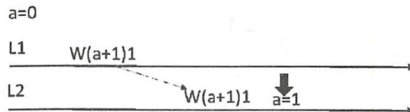


图 7-17 不满足单调写一致性

注意，单调写一致性跟以数据为中心的 FIFO 一致性类似，但是它们的使用场景不同，FIFO 一致性是多个进程同时去写，而此处强调的是针对一个进程。在实际业务中，例如有一款游戏，用户打怪升级，杀死一个怪物增加 100 经验值，此时如果不满足单调写一致性，有可能导致用户杀死怪物后，经验值没有增加的问题。

### 3. 写后读一致性（Read-your-writes Consistency）

写后读一致性是指一个进程对数据项  $a$  执行一次写操作的结果总是会被该进程对  $a$  执行的后续读操作看见。

在图 7-18 中，如果变量  $a$  的初始值为 0，那么一个进程  $P$  在  $L1$  副本上做加 1 操作，得到结果  $a=1$ ，在  $a=1$  从  $L1$  副本传递到  $L2$  之后，进程  $P$  在  $L2$  上读到了最新的值，此过程满足写后读一致性。而在图 7-19 中，读并没有得到最新的结果，导致不满足写后读一致性。

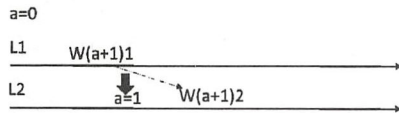


图 7-18 满足写后读一致性

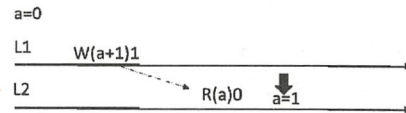


图 7-19 不满足写后读一致性

例如一个用户发了一条微博，如果后端 MySQL 采用 Master-Slave 结构做读写分离，当并发量比较大时产生了延迟，结果他没有在自己的微博列表中看到刚刚发过的微博，这就产生了不好的用户体验。通常会规定在写后  $t$  时间内读取 Master 的数据， $t$  大于 MySQL 的主从延迟时间。

### 4. 读后写一致性（Writes-follow-reads Consistency）

读后写一致性是指同一进程对数据项  $a$  执行的读操作之后的写操作，保证发生在与  $a$  读取值相同或更新的值上。

在图 7-20 中，如果变量  $a$  的初始值为 0，一个进程  $P$  在  $L1$  副本上做加 1 操作，得到结果  $a=1$ ，在  $a=1$  从  $L1$  副本传递到  $L2$  之后，进程  $P$  在  $L1$  上读到了最新的值，然后在  $L2$  上对  $a$  加 1 操作，此时的  $a$  为 2，此过程满足读后写一致性。而图 7-21 中，由于  $L2$  没有及时同步  $a$  的变化，导致进程  $P$  在  $L2$  对  $a$  加 1 的结果为 1，不满足读后写一致性。

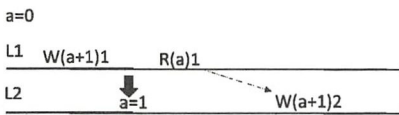


图 7-20 满足读后写一致性

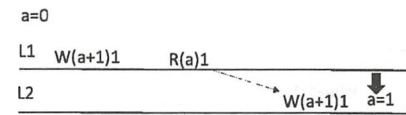


图 7-21 不满足读后写一致性

在实际业务中，用户 A 在聊天室发了一条消息，用户 B 看到了，注意这里如果副本间没有及时同步，用户 B 进行回复，这次请求路由到了另外一个副本，结果导致用户 B 的回复还在，但用户 A 的消息却没有了，用户体验不好。这种情况还不如让用户 B 更晚看到用户 A 的消息的用户体验好。

7.3.3 业界常用的一致性模型

上面主要从学术角度进行分类，在实际情况中，大家简化了分类，主要分成如下三类。

- **弱一致性**：写入一个数据 a 成功后，在数据副本上可能读出来，也可能读不出来。不能保证每个副本的数据一定是一致的。
- **最终一致性 (Eventual Consistency)**：写入一个数据 a 成功后，在其他副本有可能读不到 a 的最新值，但在某个时间窗口之后保证最终能读到。可以把它看作弱一致性的一个特例，这里面的重点是这个时间窗口。在读多写少的场景中，例如 CDN，读写比非常悬殊。假如网站的运营人员修改了一张图片，最终用户延迟一段时间才看到这个更新实际上问题不大。我们把这种一致性归结为最终一致性。最终一致性是指如果更新的间隔时间比较长，那么所有的副本最终能够达到一致性。
- **强一致性 (Strong Consistency)**：数据 a 一旦写入成功，在任意副本任意时刻都能读到 a 的最新值。

最终一致性还可以继续细分，大家更喜欢把除弱一致性和强一致性以外的其他一致性全部归为最终一致性。

弱一致性和最终一致性的副本同步采用的是异步方式，而强一致性一般要求同步更新副本后，才能返回成功，否则很难满足任意副本任意时刻都能读到最新值。异步通常意味着更大的吞吐量，但也意味着更复杂的架构、开发、调试。同步意味着简单，但也意味着响应时间更长，吞吐量更低。

表 7-1 列出了主备、主从、多主、两阶段提交 (2PC, 2 Phase Commit)、Paxos 分别在一致性、事务、响应时间、吞吐量、数据丢失、故障转移等方面的表现，可以根据场景进行选择。

表 7-1 对照表<sup>①</sup>

	主备	主从	多主	两阶段提交	Paxos
一致性	弱一致性	最终一致性	最终一致性	强一致性	强一致性
事务	无	全局事务	本地事务	全局事务	全局事务
响应时间	低	低	低	高	高
吞吐量	高	高	高	低	中
数据丢失	经常	少量	少量	无	无
故障转移	不能读写	只读	读写	读写	读写

- 主备通常不会用在生产环境。

<sup>①</sup> 翻译自 Google App Engine 的 Ryan Barrett 在 2009 年的 Google I/O 上所做的题为 “Transaction Across DataCenter” 的演讲。



- 主从可以异步，也可以同步，写是瓶颈点。
- 多主可以解决写的问题，复杂度在于如何解决冲突。
- 两阶段提交是强一致性，性能低，容易死锁。
- Paxos 是完全分布的，没有单一的主协调员，实现起来比较复杂。

从表 7-1 中可以看出，如果要满足强一致性，则可以采用 2PC 和 Paxos 两种方案，但是它们都有各自的问题。通常在满足业务场景的情况下选择最终一致性，例如聊天室只要保证最终一致性下的顺序一致性即可。

## 7.4 如何实现强一致性

### 7.4.1 两阶段提交

两阶段提交示例，如图 7-22 所示。如果让 Service 同时向 DB1、DB2 写数据，那么如何保证同时成功或者同时失败呢？

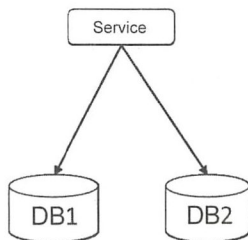


图 7-22 两阶段提交示例

这时 Service 服务需要充当协调者，指挥所有参与节点共同完成一次事务。因此，两阶段提交的思路可以概括为：由协调者决策是提交还是取消，第一阶段，协调者发送请求给参与者，参与者返回自己的建议；第二阶段，协调者发出提交或取消指令，参与者执行。

两阶段提交（如图 7-23 所示）的步骤如下。

#### （1）第一阶段

- 1) 协调者发起请求，询问参与者是否可以提交。
- 2) 参与者 1 锁定数据，返回可以提交。
- 3) 参与者 2 锁定数据，返回可以提交。

#### （2）第二阶段

- 1) 协调者发起请求，命令提交。
- 2) 参与者 1 返回提交成功。

3) 参与者 2 返回提交成功。

在这个过程中，如果任何一个参与者返回不能提交，则协调者最后会发出取消请求。

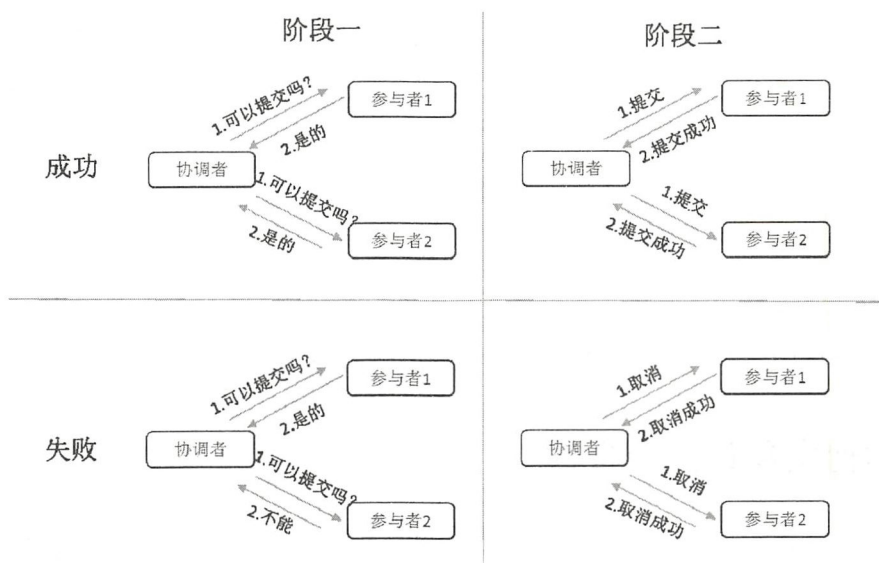


图 7-23 两阶段提交

两阶段提交存在的问题如下。

- 协调者的可用性无法保证。第一阶段锁定数据之后，如果协调者失败了，将没有角色去指挥参与者是否应该提交，这个数据会一直被锁定，那么将如何保证协调者的可用性。
- 会出现不一致。如果第二阶段参与者 1 提交成功，参与者 2 提交失败了（网断了或没返回 ACK），此时协调者不知道该如何处理。因为参与者 1 已经提交成功，外部可以访问了，注意这里出现了不一致，所以 2PC 也可能会导致不一致。
- 可扩展性极差。

#### 7.4.2 三阶段提交（3PC）

既然两阶段提交存在很多问题，那么是否还能更进一步呢？三阶段提交在两阶段提交的基础上主要从两方面进行了改进，如图 7-24 所示。三阶段提交主要做了两项改进，首先，把第一阶段拆分为两个阶段，等所有参与者都同意了再去锁定资源。这样就降低了两阶段提交提前锁定资源导致问题的概率。另外，参与者和协调者都引入了超时机制。

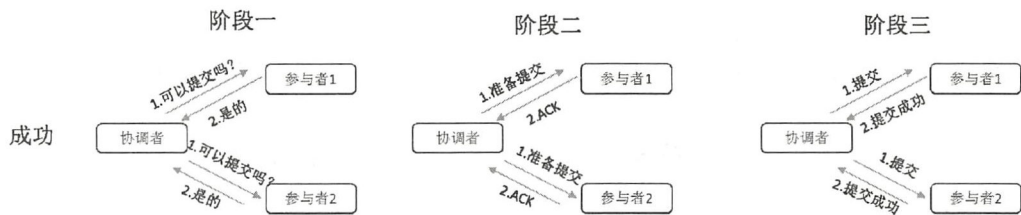


图 7-24 三阶段提交

三阶段提交同样问题较多，主要问题如下。

- 三阶段提交实现的难度更大。
- 性能也更低。
- 可扩展性极差。

特别是性能的问题，因此生产环境用三阶段提交方案的并不多。

## 7.5 如何实现最终一致性

很明显，实现强一致性并不容易，即使是两阶段提交、三阶段提交也无法保证绝对的强一致性。我们不能因为极小的不一致性概率导致性能低下，扩展性受到限制，架构变得极其复杂。因此，在业界，两阶段提交、三阶段提交缺乏大规模应用的案例，最终一致性是一个折中的方案，被大量采用。

### 7.5.1 重试机制

当 Service M 同时调用 Service A 和 Service B 时，如果调用 Service A 成功，调用 Service B 失败，那么为了保证最终一致性，最简单的办法当然是重试，如图 7-25 所示。

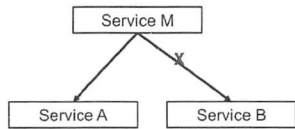


图 7-25 调用失败

重试的实现方法非常简单，但是需要注意几个问题。

一个问题是，如果 Service B 不能返回成功或失败的状态怎么办？为了避免资源耗尽、卡死，需要在 Service M 设置超时时间，超时后认为失败，继续重试，这里要设定的参数如下。



- 超时时间。
- 重试的次数。
- 重试的间隔时间。
- 重试间隔时间的衰减度。

还有一个问题曾经在阿里开源的服务化框架 Dubbo 中遇到过，那就是如果服务的调用链条比较长，出现多级重试的时候，会有叠加效应，最终导致自己被自己压死。因为虽然客户端认为超时，但是服务端的请求还会继续执行，并没有被中断。

### 7.5.2 本地记录日志

如果还没有来得及重试 Service M 就挂掉了怎么办？

一种办法是，在本地记录日志，如在调用 Service A 和 Service B 之前，记录“TranID-A-B-detail”，TranID 为事务 ID，可以生成一个随机序列号，detail 为数据的详细内容，如果调用 A 成功了，则记录“A success”。如果 Service M 出现了故障，重新启动时发现本地日志缺少“B success”，就重新调用 B，则可以定期检测并删除日志。

但是这给 Service M 带来了状态，当没有状态的时候，如果 Service M 部署在物理机 X 上，当 Service M 出现故障时，可以在物理机 Y 上创建新的节点。我们不需要在出现故障的节点上修复，而是通过另外一个新节点替换老节点即可，因为状态是复杂的。

另外一种改进方案是，本地写日志，然后收集到分布式文件系统中，启动一个检查工具不断去验证，如图 7-26 所示。当然，这种方案是为了让 Service M 去状态，本身 Service M 也可以不断检测，所以大多数问题在 Service M 中已经解决了，这是一个补充方案。因为本身这种情况发生的概率就比较小（跟数据量有关），所以也可以选择人工处理。

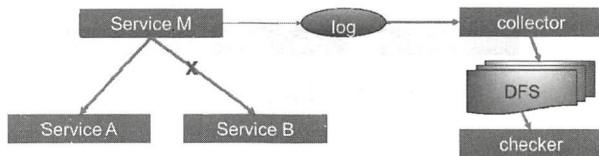


图 7-26 收集失败日志

### 7.5.3 可靠事件模式

一种观点认为，发生故障毕竟是小概率的，如果对一致性要求没有那么多高，则可以考虑这种方案，如图 7-27 所示。Service M 同时调用 Service B 和 Service A，如果调用 Service B 失败，则回去重试，重试一定的次数仍然失败，则直接发送消息给 MQ，转换为异步。这

里建议采用分布式能力比较强的 MQ，因为在某些场景下，MQ 会收到大量消息，Service B 中可以专门集成一个错误处理的组件，不断从 MQ 收集补偿消息。

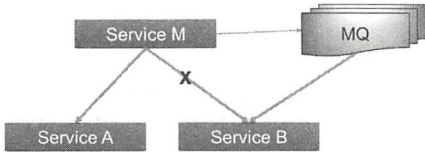


图 7-27 基于 MQ 重试

当然，这种方案也有丢消息的风险，就是 Service M 的消息没有发出来之前丢了，虽然概率比较小。

那么，有没有一种方案可以做到真正的最终一致性呢？再回过头来看看最终一致性的要求是什么——要么全部成功，要么全部失败。

另外一种方案就是可靠事件模式，如图 7-28 所示。Service M 发送一个事件到 Service A、Service B、Service C，注意这里是一个请求，这个请求肯定是要么全部成功，要么全部失败的，不存在部分成功或部分失败的情况。到了 MQ 消费数据的阶段，如果 Service C 消费失败了，则不会提交消费成功的状态，只要不断地去消费就可以了，即使 Service C 在另外一个节点重新启动也不会丢数据。因为本身 MQ 可以采用分布式 MQ，并且可以持久化，这里通过 MQ 保障不丢消息，认为 MQ 是可靠的。当 Service M 发送事件到 MQ 后就可以认为写入 Service A、Service B、Service C 成功。

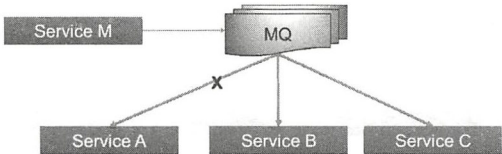


图 7-28 可靠事件模式

可靠事件模式的优势如下：

- 提升了吞吐量。
- 在某种场景下降低了响应时间。

可靠事件模式存在的问题如下：

- 存在不一致的时间窗口。
- 增加了架构的复杂度。
- 消费者需要保证幂等性。

这种方案在实际的业务中应用还是比较广泛的。这种方案是不是就完美了呢？当然不是。这里面有个问题可能是致命的，就是不一致的时间窗口。举个例子，如果创建一个订单之后，只是写入了 MQ，没有写入数据库，则可能会读不到，导致后续操作的很多不便，这里订单是一个核心业务，而其他的服务可以认为是从属业务。

如何保证写后读一致性呢？图 7-29 提供的解决方案就是，先在本地创建订单并且向 MQ 发送消息，其他业务从 MQ 消费消息，保证写后一定能够读到。如果仔细研究，你就会发现这个过程并不完美，因为发消息和更新数据库需要一个事务保护。此时，我们可以在订单库中加入事件表，把分布式事务变成单库事务。

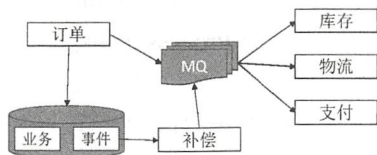


图 7-29 补偿模式

整个发送流程如下。

- (1) 创建订单，持久化到业务表并在事件表中插入一条事件信息。注意，这是在一个事务中完成的，可以保证一致性，如果失败了无须关心业务服务回退，如果成功则继续。
- (2) 发送消息，如果失败，就进行重试，如果在重试前挂掉了，由补偿服务去重新发送，补偿服务会不断地轮询事件表，找出异常的事件进行重试，如果成功则继续。
- (3) 如果成功，删除事件表中的事件信息。

有人可能会说，既然有补偿保证重发了，那么还在订单服务里面发什么消息呢，等补偿服务发送不就行了吗？注意，这里第二步直接发送消息是要增加实时性的，只有异常的时候才使用补偿服务发消息。当然，对实时性没有要求的情况下，也可以去掉实时性直接发送。

这里面额外引入了一张事件表，数据库会有额外的压力。

#### 7.5.4 Saga 事务模型

我们先来看一个例子，电商系统业务逻辑比较复杂，我们通常采用微服务架构，如果进行下单操作，通常会伴随着多个子任务（大型电商有几百个子任务），比如扣减优惠券、减掉库存等。如果这个例子用 Saga 来实现一致性，可以由一个 Process manager 统一串接起来，如图 7-30 所示。



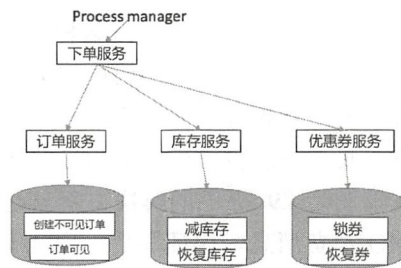


图 7-30 Saga 事务模型

这就是传说中的 Saga 事务模型<sup>①</sup>，又叫 Long-running-transaction，核心思想是把一个长事务拆分为多个本地事务来实现，由一个 Process manager 统一协调。如果成功则继续往下执行，如果失败则调用补偿操作。每个业务都至少要实现正向、负向两个接口，以前创建订单的时候只要一次操作就可以了，现在需要两次，第一次创建不可见的订单，等到所有操作都结束，才让订单可见。下单流程，如图 7-31 所示。

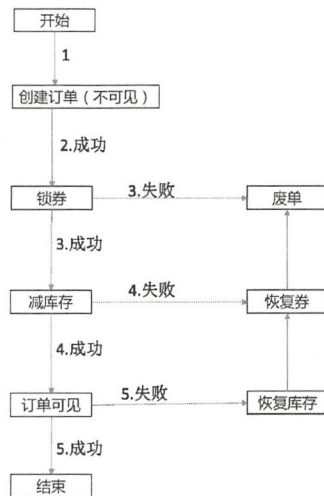


图 7-31 下单流程

如果仔细研究会发现，执行到第 5 步时如果失败，回退的过程也比较复杂，特别是子业务比较多时的时候。

此处有三种方式去保证。

- 建立一个定时任务去检查数据完整性，例如执行到第 5 步失败了，定时任务会检测

<sup>①</sup> 它是由普林斯顿大学的 H.Garcia-Molina 等人提出的。

到并修复数据。

- 如果失败则发送消息到 MQ，消费者根据状态去做回退操作。同时也可以检测数据库中数据的完整性。
- 所有的操作在正向调用的时候就写好回退的日志，此日志要求持久化，可以选择用本地文件、数据库等存储。这也是华为开源微服务框架 ServiceComb 的做法。

Saga 常常在 CQRS 模式中提起，这个方案并不完美，要求幂等，并且有不一致的可能，但是实现起来并不是特别复杂。一旦出现不一致可以人工干预，毕竟出现不一致的概率非常低。这种方案在实际中应用还是比较广泛的。

### 7.5.5 TCC 事务模型

两阶段提交是依赖于数据库提供的事务机制，再配合外部的资源协调器来实现分布式事务。这种做法的问题是反伸缩性，性能比较差。当业务达到一定规模的时候，业务服务通过状态外置，增加实例个数，比较容易扩展，而数据库的扩展涉及迁移数据等问题，往往成为系统的瓶颈。因此，我们通常会想方设法降低数据库的压力，从而降低数据库扩展带来的复杂性。TCC（Try Confirm Cancel）事务模型의思想和两阶段提交虽然类似，但是却把相关的操作从数据库提到业务中，以此降低数据库的压力，并且不需要加锁，性能也得到了提升。TCC 事务模型，如图 7-32 所示。

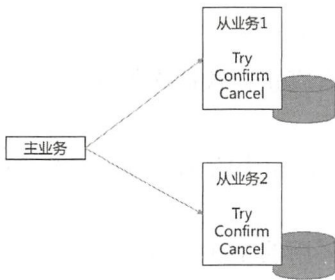


图 7-32 TCC 事务模型

TCC 是下面三个单词的缩写。

- Try: 检查业务一致性，预留业务资源。
- Confirm: 确认执行业务操作。
- Cancel: 取消执行业务操作。

TCC 事务的步骤大致如下。

(1) 主业务发起事务，分别调用从业务 Try 接口，检测业务数据一致性，预留资源。

例如优惠券服务，在数据库中增加一列，锁定优惠券，使它不能被重复使用，这里相当于两阶段提交中的加锁操作，数据不能在 Try 到 Confirm 这段时间发生变化，因为变化会导致 Confirm 失败。

(2) 如果 Try 返回成功，则分别调用从业务 Confirm 接口，提交数据。注意，这里不会再检查业务数据了，只能使用上一步中已经预留的资源。例如优惠券服务，此阶段可以使用 Try 阶段锁定的优惠券，直接把优惠券扣掉。

(3) 如果 Try 从业务的任意一个返回失败或者超时，都会分别调用 Cancel 接口，将前面预留的资源释放。

在这个过程中，我们发现有两个问题可能会发生。

第一个问题，主业务充当协调角色，和两阶段提交类似，如果主业务已经 Try 成功，并且冻结了资源，那么在执行 Confirm 的时候，主业务挂掉，将会导致不一致。

这个问题的解决方案是建立一个定时任务去检查从业务的数据完整性，定时任务会检测并修复数据。另外，主业务服务的所有操作日志也可以记录下来，定时检查不一致。

第二个问题，在 Confirm 阶段，如果从业务 1 已经提交成功，而此时从业务 2 提交失败了，如网断了，该如何处理？此时，外部已经读到了从业务 1 的数据。

这个问题我们只能做到最终一致性，两阶段提交也无法解决短时间的不一致，但是发生的概率比较小。

TCC 的优势如下。

- 在业务层处理，平衡数据库的压力。
- 比 2PC 性能好很多，没有真正在数据库加锁。

TCC 的问题如下。

- 增加业务复杂度，需要提供相应的 Try、Confirm、Cancel 接口。
- 需要提供幂等性接口。

支付宝目前的 XTS 框架就采用的 TCC 模式。

## 7.6 分布式锁

为什么需要锁？在分布式场景中，任何操作都可能跨网络，在并发场景下，如果无法保证顺序性，则会产生冲突。分布式系统的复杂之处在于，在不同进程需要互斥地访问共享资源时的的问题。

典型的冲突如下。

- 丢失更新：一个事务的更新覆盖了其他事务的更新结果，就是所谓的更新丢失。
- 脏读：当一个事务读取其他完成一半事务的记录时，就会发生脏读取。



为了解决这些并发带来的问题，我们需要引入并发控制机制。

### 7.6.1 基于数据库实现悲观锁和乐观锁

基于数据库可以实现两种加锁方式，一种是悲观锁，一种是乐观锁。

#### 悲观锁

悲观锁实际上是利用了数据库的锁机制，因此并不是所有数据库都可以使用悲观锁，例如 Hbase 数据库本身并没有锁机制，因此无法实现悲观锁。悲观锁的初衷是假设发生并发冲突，通过锁屏蔽一切可能违反数据完整性的操作。

以下是一个典型的悲观锁调用。

```
select * from account where id="xxx" for update
```

这条 sql 语句锁定了 account 表中所有符合检索条件（id="xxx"）的记录，因此检索条件范围越小越好，最好是唯一键，否则会导致并发量非常低，引起很严重的事故。一旦锁定，本次事务提交之前（事务提交时会释放事务过程中的锁），外界无法修改这些记录。

悲观锁的性能取决于锁的范围及执行时间，当使用唯一键时，MySQL 一般吞吐量在几百 TPS，如果使用 SSD，则吞吐量至少会提升一倍。

#### 乐观锁

乐观锁并没有利用数据库的锁机制，而是在业务侧进行控制。乐观锁假设不会发生并发冲突，只在提交操作时检查是否违反数据完整性。乐观锁不能解决脏读的问题。

乐观锁假设数据一般情况下不会造成冲突，所以在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测。如果发现冲突了，则给用户返回错误信息，让用户决定如何去做。

乐观锁通常是通过在数据库的表中增加一个额外字段来实现的，例如 version 字段。当读取数据时，将 version 字段的值一同读出，数据每更新一次，对此 version 字段的值加 1。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的 version 字段的值是否相等，如果数据库表当前版本号与第一次取出来的 version 字段的值相等，则予以更新，否则认为是过期数据。

当没有冲突的时候，如图 7-33 所示，详细步骤如下。

(1) 进程 1 读取数据，sql 为 `select * from account where id="xxx"`，此时读取到的 version 字段的值为 1。

(2) 进程 1 完成业务操作后，写入数据，同时更新 version 字段的值为 2，sql 为 `update account set name='xxx',version=2 where version=1`，执行成功。

(3) 进程 2 读取数据，读取到的 version 字段的值此时为 2。

(4) 完成业务操作后，进程 2 写入数据，同时更新 version 字段的值为 3，sql 为 `update account set name='xxx',version=3 where version=2`，执行成功。

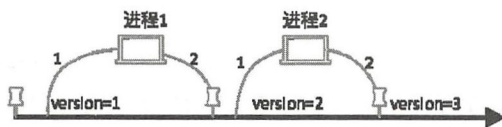


图 7-33 乐观锁-无冲突

当存在冲突的时候，如图 7-34 所示，详细步骤如下。

(1) 进程 1 读取数据，sql 为 `select * from account where id="xxx"`，此时读取到 version 字段的值为 1。

(2) 进程 2 读取数据，由于进程 1 还没有更新 version 字段的值，进程 2 读取到 version 字段的值此时也为 1。

(3) 进程 1 完成业务操作后，写入数据，同时更新 version 字段的值为 2，sql 为 `update account set name='xxx',version=2 where version=1`，执行成功。

(4) 进程 2 完成业务操作后，写入数据，同时更新 version 字段的值为 2，sql 为 `update account set name='xxx',version=2 where version=1`，执行失败。

(5) 进程 2 重新读取数据，读取到的 version 字段的值此时为 2。

(6) 进程 2 完成业务操作后，写入数据，同时更新 version 字段的值为 3，sql 为 `update account set name='xxx',version=3 where version=2`，执行成功。

如果还是有冲突，则需要不断重复，直到成功为止。

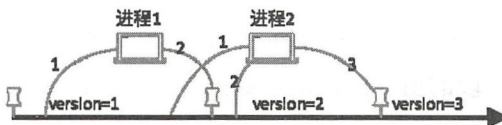


图 7-34 乐观锁-有冲突

两种锁各有优缺点，不可认为一种好于另一种。乐观锁适用于写比较少的情况下，即冲突真的很少发生的时候，这样可以省去锁的开销，加大系统的整体吞吐量。但如果经常产生冲突，上层应用会不断重试，这样反倒降低了性能，所以后一种情况下悲观锁的效果比乐观锁的效果好。



## 7.6.2 基于 ZooKeeper 的分布式锁

ZooKeeper 是一个分布式协调服务，由雅虎创建，是 Google 分布式协调服务 Chubby 的开源实现。ZooKeeper 常用于维护配置信息、注册发现服务、集群管理及分布式锁。

在 ZooKeeper 集群中，节点共有如下三种角色。

- Leader。
- Follower。
- Observer。

一个 ZooKeeper 集群同一时刻只能有一个 Leader，当 Leader 不可用后会从 Follower 中重新选举一个 Leader 出来，以此来保证 ZooKeeper 集群的高可用。

在 ZooKeeper 中，节点类型可以分为持久节点（PERSISTENT）、临时节点（EPHEMERAL）及时序节点（SEQUENTIAL），通过组合可以产生如下四种节点类型。

- 持久节点，在 ZooKeeper 与客户端断开连接后，该节点依旧存在。
- 持久顺序节点（PERSISTENT\_SEQUENTIAL），节点名称有顺序编号，ZooKeeper 与客户端断开连接后，该节点依旧存在。
- 临时节点，ZooKeeper 与客户端断开连接后，该节点被删除。
- 临时顺序节点（EPHEMERAL\_SEQUENTIAL），节点名称有顺序编号，ZooKeeper 与客户端断开连接后，该节点被删除。

通过 ZooKeeper 实现分布式锁的思路，就是通过创建临时顺序节点来实现的。

(1) 创建一个 PERSISTENT 类型的 znode，/Locks/write\_lock。

(2) 客户端创建 SEQUENCE|EPHEMERAL 类型的 znode，名字以 lockid 开头，创建的 znode 是 /Locks/write\_lock/lockid0000000001。

(3) 调用 getChildren()，不要设置 Watcher 获取 /Locks/write\_lock 下的 znode 列表。

(4) 判断步骤 (2) 创建的 znode 是不是 znode 列表中最小的一个，如果是就代表获得了锁，如果不是就继续。

(5) 调用 exists() 判断步骤 (2) 创建的节点编号 1 的 znode 节点（也就是获取的 znode 列表中最小的 znode），并且设置 Watcher。如果 exists() 返回 false，则执行步骤 (3)。

(6) 如果 exists() 返回 true，那么等待 zk 通知，从而在回调函数里返回执行步骤 (3)。

(7) 释放锁就是删除 znode 节点或者断开连接。

需要注意的是，步骤 (2) 中 getChildren() 不设置 Watcher 的原因是防止“羊群效应”。如果 getChildren() 设置了 Watcher，那么集群每次一抖动都会收到通知。在整个分布式锁的竞争过程中，大量重复运行，并且绝大多数的运行结果都是判断出自己并非是序号最小的节点，从而继续等待下一次通知。如果客户端接收到过多的和自己不相关的事件通知，在





集群规模大的时候，会对 Server 造成很大的性能影响，并且一旦同一时间有多个节点的客户端断开连接，这个时候，服务器就会像其余客户端发送大量的事件通知——这就是所谓的“羊群效应”。

### 7.6.3 基于 Redis 实现分布式锁

基于 ZooKeeper 实现的分布式锁毕竟吞吐量有限，而基于 Redis 实现的分布式锁的吞吐量至少可以提升一个数量级。

#### Redis 单节点方式实现

利用 Redis 实现分布式锁，最简单的是单节点方式。通过单点实现分布式锁的核心是围绕 SETNX（SET if Not eXists）展开的，当 key 不存在时返回 1，当 key 存在时返回 0。SETNX 的执行演示，如图 7-35 所示。

```
10.1.1.1:6380> setnx dog0 1
(integer) 1
10.1.1.1:6380> setnx dog0 1
(integer) 0
```

图 7-35 SETNX 的执行演示

因为 Redis 是单线程的，所以在 Redis 服务侧不会有线程安全问题。当返回 1 的时候认为获得锁成功，可以进行相应的业务处理，处理完成后，删除 key，释放锁；当返回 0 时表示失败。

很明显，这里存在很多问题。

#### （1）超时问题如何解决？

如果线程 A 拿到了锁，在处理业务的过程中发生阻塞，例如数据库执行比较慢，或者 Service1 发生故障了，这时候如果没有超时限制，系统将永久锁死，所以 SETNX 可以接受第三个参数，也就是超时时间。超时问题，如图 7-36 所示。

这里面存在问题，因为你并不知在超时的情况下，业务到底有没有处理成功，还有没有在继续进行，只是客户端认为超时了，服务端并不知道，所以这里的锁并不是绝对的。

#### （2）如何释放锁？

直接删除可以吗？答案是否定的。这里要注意另一个问题，因为超时时间是放在 Redis 服务端计算的，如果 Service 1 超时了，但是它自己是不知道自己超时的，除非不断轮询 Redis 确认，不断轮询也是有问题的，因为轮询是有时间差的。例如，你请求 Redis 的时候还没超时，删除的时候恰好超时了，Service 2 刚拿到锁，Service 1 就误删了 Service 2 的锁，造成锁失效。



解决方案就是，使用 SETNX 的时候，value 值可以在客户端生成一个随机值，例如 `set lock_name random_value`。删除的时候根据 key 获取 value，如果相同就删除。当然，这个地方必须是原子的，否则判断到删除之前还是有可能发生变化的。我们可以通过 Lua 脚本来实现。

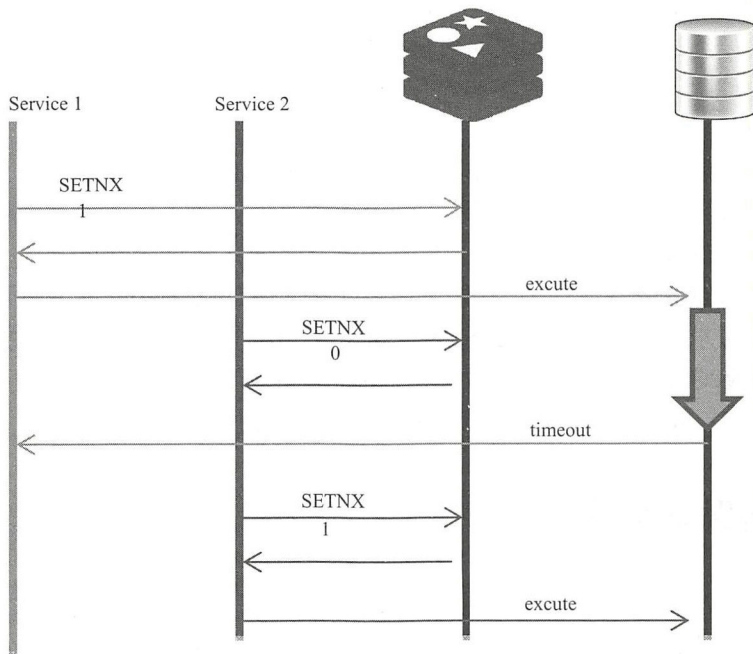


图 7-36 超时问题

### (3) 单点问题如何解决？

还有另外一个问题，Redis 是单点的，如果 Redis 挂掉了，那么整个系统就会全部崩溃。

有人认为可以用 Master-Slave，但是 Master-Slave 之间是异步传输数据的，也就是不能设定为 Master 和 Slave 都写成功了才返回。Redis-Cluster 也是异步的。

## Redlock 实现方案

Redlock 是 Redis 的作者 Antirez 在 Redis 官网中给出的一种基于 Redis 的分布式锁方案。直白点说，就是  $N$ （通常是 5）个独立的 Redis 节点同时使用 SETNX。如果多数节点成功，就拿到了锁，这样就可以允许少数（如 2 个）节点不可用。整个取锁、释放锁的操作和单节点类似。

是不是这样就完美了呢？当然不是。



### (1) 重启问题导致锁失效。

假设一共有 5 个节点 (A/B/C/D/E)，Service 1 成功获取了锁，注意这里 Service 1 在 A/B/C 上 SETNX 成功，但是并没有在 D/E 上成功。假如 C 节点挂掉后又恰巧重启了，C 节点并没有持久化，这时候 Service 2 也可能已经锁住 C/D/E，导致锁失效。

有人说，那 C 持久化不就行了吗？实际上设置为同步的持久化方式对性能影响比较大，也就是常说的始终追加同步，如果是机械硬盘，吞吐量可能会从 MB 级降低到 KB 级。有人说用固态硬盘不就解决问题了吗？使用固态硬盘，吞吐量确实能到万级，但是大量而频繁地写入容易导致写入放大。

这个问题的解决方案也非常简单，就是延迟重启 (Delayed Restarts)，即等到挂掉节点上的所有锁都过期后再重启。重启后，以前的锁都已经失效了，参考租约机制原理。

### (2) 任何时候都需要全部删除。

如果一个节点获取锁成功了，那么四个节点都 SETNX 成功。一个失败了，失败的情况如果是发起请求成功，在返回 ACK 的时候失败，这时客户端会认为是失败了，而删除锁的时候没有删除这个节点，这就导致过期之前，这个节点获取锁一直失败，所以正确的做法是无论 SETNX 成功还是失败，都应该执行一次删除操作。

## 7.7 如何保证幂等性

上面我们多次提到了幂等性，那么什么是幂等性呢？幂等性是指一次和多次请求某一个资源具有同样的作用。如果用公式，则可以表达为  $f(x)=f(f(x))$ 。

用例子来说明一下：把编号为 5 的记录的 A 字段设置为 0，这种操作不管执行多少次都是幂等性的；而把编号为 5 的记录的 A 字段增加 1，这种操作显然就不是幂等性的。

### 7.7.1 幂等令牌 (Idempotency Key)

客户端 (Client) 调用服务端 (Server) 幂等性，如图 7-37 所示。假设 Server 对外提供了一个接口，Client 通过 request 调用 Server，如果要保证请求一定被处理成功，就要通过 response 来通知 Client，但是如果超时了，那么到底是成功了，还是失败了，Client 是不知道的。

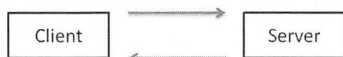


图 7-37 客户端调用服务端幂等性

有的人可能会说，去查询一次处理结果，但是这里存在如下两个问题。





- 可能查不到处理结果，例如对某个字段减 1，则无法查询。
- 可能查询的时候还没处理成功，发起重试的时候却成功了，如在 CQRS 模式中，更新采用 MQ 异步，读取从缓存中获得，很可能存在延迟。

一旦请求失败进行重试的时候，Server 端是不知道这个请求是否被处理过的。因此，如果为了保证 Server 端能够识别同一个请求，必须要有一个令牌标识，在请求的过程中一起发送给 Server 端，Server 端利用令牌来识别是不是同一个请求。这里有很多解决方案，总之，要把这个令牌存储起来，在下次请求过来的时候进行比对。

图 7-38 是基于 Redis 来实现幂等性的方案，在 Redis 中，用令牌 Token 作为 key，value 有两个，1 代表正在处理，2 代表已经处理成功。

(1) Client 发起请求，携带令牌 Token。

(2) Server 在收到请求的时候，先去 Redis 查询令牌。如果查不到，则通过 SET 命令保存 Token。这里需要设置过期时间。如果查询到的结果为 1，则休眠一段时间重试；如果查询到的结果为 2，则返回成功。

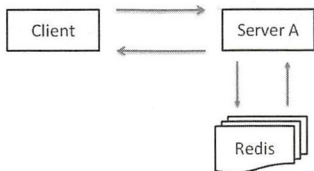


图 7-38 基于 Redis 实现幂等性

这里面存在一个问题，通过 SET 命令可能会有并发性问题，即可能存在两个线程同时处理一个请求的情况。你可能会说，幂等性是针对一个请求的，不会有第二个线程也处理同样的数据，但是在下面这种场景中则会出现两个线程同时处理一个请求的状况。Client 发送请求到 Server，Server 发生了 FullGC，出现了超时，此时 Client 发起重试，Server 的 FullGC 结束，此时就会有两个线程处理相同 Token 的数据。

要解决这个问题也不难，将 SET 换成 SETNX 命令保存在 Token 中，如果返回 1，则继续处理；如果返回 0，说明有另外的线程抢先了，则重新查询。

很多初学者可能非常容易把幂等性和并发性弄混。要解决幂等性问题，首先要解决并发性问题，但是解决了并发性问题还不能解决幂等性问题。加锁是为了解决并发性问题，但是加了锁也不一定能解决幂等性问题。我们来举例说明，通过乐观锁可以解决并发性问题，但是加了乐观锁就一定幂等了吗？答案是不一定。

另外一个问题，幂等在每一次分布调用都会涉及。幂等问题的传递性，如图 7-39 所示。就算前面保证了幂等性，后续调用仍会面临幂等性问题。不到最后一个环节，就不能说完全解决了幂等性问题。因此，一般来说，在数据库一侧实现幂等更有效。



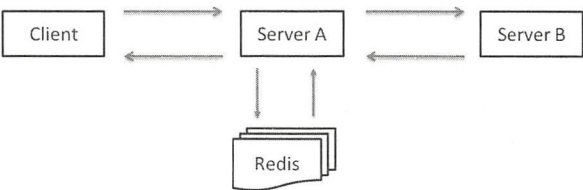


图 7-39 幂等问题的传递性

7.7.2 在数据库中实现幂等性

在图 7-40 中，无论前面幂等性实现的如何，到了这个环节还是一样，要保证 Service 的数据被写到 DB 中，这通常通过 ACK 机制来实现。但是如果 request 成功，response 失败，则这时候我们会认为整个执行失败，但是事实有可能是执行成功后在 response 的时候才发生断网，这时候我们必然选择重试，最终可能会造成重复数据。

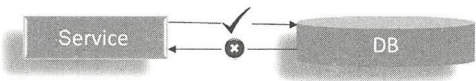


图 7-40 更新成功，返回失败问题

那么，如何解决这个问题呢？  
我们可以在一张表中建立一个唯一键，通过数据库的唯一约束保证幂等性。例如订单表，订单 ID 是全局唯一的，在订单表有唯一约束，如果重复下订单，则肯定会抛出异常。  
有人会说，并不是每个表都有唯一约束，如果是库存表（如表 7-2 所示），库存只是一个数量，每次请求对库存减 1，那么如何建立唯一约束呢？

表 7-2 库存表

商品 ID	库存	状态	.....
1101	99	1	.....

我们可以利用唯一键增加一张流水表，如表 7-3 所示。在库存流水表中，可以以订单 ID 作为唯一约束，记录库存增减流水，减库存的同时更新两张表，并且作为一个单库事务。

表 7-3 库存流水表

订单 ID	商品 ID	库存	.....
112323	1101	-1	.....



# 8

## 第 8 章 未来值得关注的方向

几年前，在技术大会上热门的技术还是 OpenStack，现在已经变成了 Docker；几年前在技术大会上热门的大数据如今也变成了人工智能。在架构领域，Serverless 和 Service Mesh 是非常重要的两个方向。

### 8.1 Serverless

#### 8.1.1 什么是 Serverless

在早期，开发人员开发应用的时候，需要考虑很多层面的东西，包括用多少台服务器主机，占多少内存，数据库用什么，怎么扩展，消息处理用什么，怎么扩展。后来 IaaS 平台帮助我们实现了资源流动，通过 PaaS 平台帮助我们实现了公共基础服务层的抽象<sup>①</sup>。但是，开发人员要关注什么呢？开发人员仍然要考虑灰度发布、容错、并发等一系列架构问题，而 Serverless 的目标就是帮助开发人员减少对基础设施及公共基础服务的关注，包括用哪个容器，让业务开发人员更简单容易地实现业务逻辑，关注更多的架构细节。

Serverless 是一个比较新的概念，没有比较权威的定义，目前最新的一个定义是这样描述的：“无服务器架构是基于互联网的系统，它的应用开发不使用常规的服务进程。相反，

---

① 云计算的三种模式：IaaS（基础架构即服务）、PaaS（平台即服务）、SaaS（软件即服务）。





它仅依赖于第三方服务（例如 AWS Lambda 服务），是客户端逻辑和服务托管远程过程调用的组合”。

Serverless 并不是不需要服务，而是开发者不用关注服务。举个例子，开发一个应用，开发者需要关心缓存、MQ、Web 容器。而在 Serverless 环境下，开发者只需要关注代码层面的东西，也就是写完代码直接提交到 Serverless 服务上，并设置相关的策略，Serverless 就会帮开发者考虑好一切，包括基础设施、扩展性、性能等。如果你想用 MQ，则只需调用函数，无须关注 MQ 是否能承受压力，至于什么时候需要扩展，成本如何控制等问题，Serverless 会为你做好一切。

2012 年，Serverless 第一次出现在 Ken Form 写的一篇名为 *Why The Future of Software and Apps is Serverless* 的文章中。AWS 在 2014 年发布了 Lambda 让 Serverless 得到了广泛关注，为基于云的应用提供了一种全新的架构模式。2016 年 10 月在伦敦举办了第一届 Serverlessconf。在两天时间里，来自全世界 40 多位演讲嘉宾分享了 Serverless 的最新进展及下一步计划。

在 Serverless 的世界里，除了 AWS，还有很多其他厂商都在积极推行 Serverless，例如 Google 的 Cloud Functions、Microsoft 的 Azure Functions、IBM 的 OpenWhisk，以及 Iron.io 和 Webtask 等各种开源平台都提供了类似的服务。

### 8.1.2 Serverless 的现状

Serverless 虽然是各大厂商争相实验的区域，但是目前并不成熟，它还没有特别成功的案例。下面列举 Amazon、Google、Facebook 在 Serverless 领域进行的尝试。

2014 年 11 月，Amazon 发布 Lambda。Lambda 可以简单地理解为：一种可以直接根据时间来运行用户代码的计算服务，说 Lambda 是更有深度的 PaaS 也不为过，你不需要关心底层的任何存储和计算资源。

使用 Lambda 有什么好处？

Lambda 的好处是省钱。它根据函数调用次数收费，每月前 100 万个请求免费<sup>①</sup>，这里需要注意，虽然 Serverless 不收费，但是其他的费用照常，例如 API Gateway。

Lambda 能做到的是，你只需要关心业务逻辑，而无须关注架构细节，可以非常容易地实现扩展或缩容。它是如今市面上较早、较成熟、较稳定的 Serverless 框架之一。这项服务最初只支持 Node.js，现在还支持 Java 和 Python。

就在 Amazon 发布 Lambda 的前一个月，Google 收购了 Firebase，Firebase 可以简单地理解为通过 API 去构建实时性应用。它不需要额外的服务器基础设施，就可以做到随时扩

---

① 具体可以参考官方网站（<https://aws.amazon.com/cn/lambda/pricing/>）。



展或扩容，对数据库的存储也没有限制。Firebase 号称最高可以处理 MB 级的并发和 TB 级的数据传输，数据的延迟在 10 毫秒级别。Firebase 简单到什么程度？你只需要编写前端代码（HTML+CSS+JS），后端代码几乎可以不写，十几行代码就可以实现一个实时性聊天室。

另外，Google 还推出了 Google Cloud Functions，目前仍然处于初期，只支持 Node.js。

微软、阿里、华为等厂商在 Serverless 方面的反应相对比较迟缓，但是它们都是比较全面的公有云平台，这些厂商也相继推出了 Serverless 服务。

无论哪个厂商建立的 Serverless 都具有如下特点。

- 在 Serverless 架构模式下，开发人员只需要专注于业务代码实现，不需要关心运维、扩容等其他工作。
- Serverless 按需使用的粒度更小，有请求的时候才开始建立服务，运行服务。
- 基于 Serverless 开发的应用严重依赖于特定的云平台，目前还无法相互兼容，这意味着可能会被厂商绑定。
- Serverless 是按调用次数收费的。

从开发者的角度看，Serverless 具备哪些优势呢？

- 成本低。只有运行的时候才启动服务，按调用次数收费，花费更少。
- 开发速度更快。开发人员需要关注的范围更小，开发速度大幅提升。
- 质量属性高。由于云厂商的基础设施及公共基础服务都有 SLA 要求，Serverless 经过大规模验证过，质量属性相对较高。
- 简化运维。相关的运维工作全转移到云厂商一侧。

相应地，Serverless 目前也存在很多问题。

- 成功案例太少。目前的情况也只适合简单的应用开发，缺乏大型成功案例的推动。
- 很难满足个性化。跟 PaaS 平台一样，开发者的个性化需求绝对不想被绑定。
- 缺乏行业标准。在 AWS 上能运行，是否意味着被厂商绑架了，有没有一套标准可以适用所有的云。
- 初次访问性能差。这种有请求才启动服务的方式，在启动服务的这段时间，一定会降低响应时间，影响用户体验。
- 缺乏开发调试工具。在开发验证的过程中，需要不断地把代码上传到服务端运行调试。

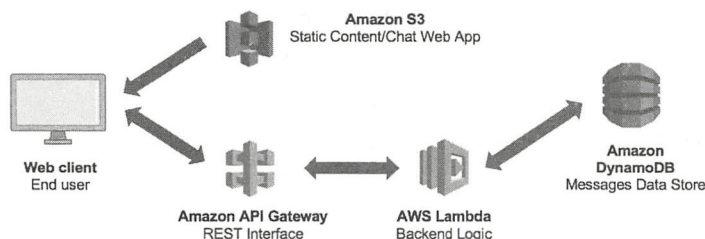
### 8.1.3 Serverless 的应用场景

以 Lambda 为例，一个简单的 Serverless 架构大概包括 API Gateway、S3、Lambda、DynamoDB 服务。Lambda 架构，如图 8-1 所示。

- API Gateway 作为服务网关，处理请求，提供 API 调用，还包括流量管理、授权和访问控制、监控等。



- Lambda 基于事件驱动进行逻辑处理，这里包含业务开发人员上传的代码。
- S3 作为静态存储，包括图片、JS、CSS 等前端静态资源存储。
- DynamoDB 作为数据库存储业务数据。

图 8-1 Lambda 架构<sup>①</sup>

就 Serverless 的特点和现状来说，Serverless 有哪些实际的应用场景呢？

- 发送消息。例如邮件、短信、推送消息，这种业务对实时性要求没有那么高，访问量波动很大，采用 Serverless 能节省很多成本。
- 定时任务。例如电商每个月月底进行结算，运行结束后就消亡，这种场景不需要考虑首次运行启动容器耗费大量时间的问题。
- 文件处理。例如异步生成图片的缩略图，定时对文件进行合并压缩等。
- 低频请求。例如物联网场景，一些设备可能几分钟上报一次数据，基于事件驱动，然后对数据进行统计分析，非常典型的 Serverless 场景。

虽然在目前 Serverless 主要的战场在公有云，但是有一种场景将来需要我们格外注意，那就是边缘计算和 Serverless 的结合，因为在一个数据中心，我们可以通过云操作系统轻松管理数万个节点，而边缘的计算能力则要小得多，通常只有 10~100 台物理机，但是这些机器需要为上万个租户提供 IaaS 服务。如果是完全隔离的环境，不共用基础设施，那么没有资源流动，根本就跑不起来；如果边缘能够提供 Serverless 环境，那么就可充分利用资源，并且可以简化边缘侧的运维管理工作。

## 8.2 Service Mesh

### 8.2.1 什么是 Service Mesh

Service Mesh 是最近才兴起的一个名词，最早在 2016 年 9 月 29 日由开发 Linkerd 的

<sup>①</sup> 图片来自 <https://aws.amazon.com/cn/blogs/compute/surviving-the-zombie-apocalypse-with-serverless-microservices/>。



Buoyant 公司提出<sup>①</sup>。2016 年 10 月, Alex Leong 开始在 Buoyant 公司的官方博客中连载一系列关于“A Service Mesh for Kubernetes”的文章。随着 2017 年 Linkerd 加入 CNCF, Service Mesh 开始大规模出现在各个技术论坛上。Service Mesh 在国内被翻译为“服务啮合层”或“服务网格”。那么到底什么是 Service Mesh 呢?目前比较公认的定义是 Buoyant 的 CEO William Morgan 在博客<sup>②</sup>中给出的,具体描述如下:

Service Mesh 是用于处理服务到服务通信的专用基础架构层。Cloud Native 有着复杂的服务拓扑,它负责可靠地传递请求。实际上,Service Mesh 通常作为一组轻量级网络代理实现,这些代理与应用程序代码部署在一起,应用程序无感知。

随着 Cloud Native 的崛起,Service Mesh 逐步发展为一个独立的基础设施层。在 Cloud Native 架构下,单个应用程序可能由数百个服务组成;每个服务可能有数千个实例;并且这些实例中的每一个实例都可能处于不断变化的状态,因为它们是由诸如 Kubernetes 之类的资源调度系统动态调度的。这个世界中的服务通信不仅非常复杂,而且是运行时行为的普遍和基本部分,管理它对于确保端到端的性能和可靠性至关重要。

Azure 的云设计模式库也描述了类似的模式,名为 Sidecar<sup>③</sup>,这个概念很形象,就是我们以前在战争影片中看到的那种挎斗车,如图 8-2 所示。

在模式库中,Sidecar 被这样描述:

将应用程序的组件部署到单独的进程或容器中以提供隔离和封装。这种模式还可以使应用程序由异构组件和技术组成。

这种模式被命名为 Sidecar,因为它类似于连接到摩托车的辅助车。在该模式中,辅助车被附加到父应用程序并为应用程序提供支持功能。辅助车也与父应用程序共享相同的生命周期,与父进程一起被创建和退出。

简单来说,Service Mesh 帮助应用程序在海量服务、复杂的架构和网络中建立稳定的通信机制,业务所有的流量都转发到 Service Mesh 的代理服务中。不仅如此,Service Mesh 还承担了微服务框架所有的功能,包括服务注册发现、负载均衡、熔断限流、认证鉴权、

---

① 2016 年 1 月 15 日,离开 Twitter 的基础设施工程师 William Morgan 和 Oliver Gould,在 GitHub 上发布了 Linkerd 0.0.7 版本,同时创建了一个创业公司 Buoyant。业界第一个 Service Mesh 项目在这个公司诞生。

② 博客地址 <https://blog.buoyant.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one/>。

③ 参见 <https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar>。

缓存加速等。不同的是，Service Mesh 强调的是通过独立的进程代理的方式，除此之外，还承担了上报日志、监控的责任。Service Mesh 架构，如图 8-3 所示。



图 8-2 现实中的 Sidecar<sup>①</sup>

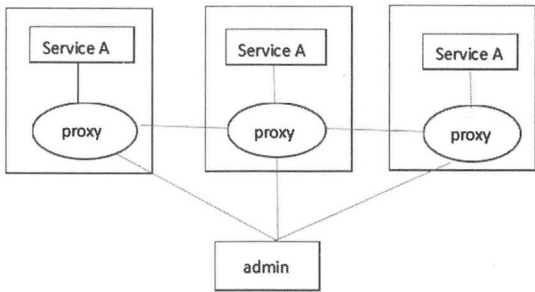


图 8-3 Service Mesh 架构

### 8.2.2 为什么需要 Service Mesh

Service Mesh 的出现是由微服务架构推动的，随着一个应用被拆分成几百个甚至几个应用，服务治理面临巨大的挑战。这个时候，微服务框架出现，例如，Finagle、Dubbo、Spring Cloud、Netflix OSS 等。这些框架都是基于客户端负载均衡直连的方式，此方案的优势是性能高、应用性好，如 Dubbo。如果你使用 Java 语言，可以直接引入 JAR 包，通过简单的配置，就可以实现微服务之间的通信。

<sup>①</sup> 图片来自 <https://www.google.com.hk/search?q=%E8%BE%B9%E4%B8%89%E8%BD%AE&newwindow=1&safe=strict&tbm=isch&tbo=u&source=univ&sa=X&ved=2ahUKEwjR4Yi03sHdAhUK9LwKHTonAAQQsAR6BAgFEAE&biw=1918&bih=866#imgrc=D3mVR6XVmpKkeM:>。

归根结底，在微服务架构中，我们要解决的问题是，让开发人员感觉不到微服务之间的通信。当服务数量越来越多，升级微服务框架变得越来越复杂的时候，你不可能要求微服务框架一直不变，而且是没有 bug 的。在技术更新速度如此之快的年代，就更不可能了。因此，微服务框架的部分功能开始逐步向服务端移动，希望客户端可以尽量“薄”，但是客户端不可能无限制的“薄”，剩余部分仍然比较“厚”。

因为使用客户端更像一种交付的模式，不容易变更，控制力较差，而 Service Mesh 则从业务进程集成客户端的方式演进为独立进程的方式，也就是说，原本的客户端变成了一个独立进程。对这个独立进程升级、运维要比绑在一起强得多。

微服务架构更强调去中心化、独立自主、跨语言，但是通常微服务框架限制了这一点，不可能为每种语言都实现一个框架，要么都用一种语言，要么实现多种语言的框架。而 Service Mesh 通过独立进程的方式进行了隔离，可以低成本实现跨语言。

随着 Docker 及 Kubernetes 的崛起，微服务的部署模式开始发生转变，越来越趋向于轻量级，越来越强调隔离自治。每个服务独立占用一个容器，将服务、依赖包、操作系统、监控运维所需的代理打包成一个镜像。这种模式促成了 Service Mesh 的发展，让 Service Mesh 实现起来更容易。否则开发人员需要额外维护 Service Mesh 进程，就非常麻烦了。

### 8.2.3 Service Mesh 的现状

目前 Service Mesh 的框架如雨后春笋般快速涌现，以下几个框架是目前为止被提到次数最多的。

#### Linkerd

2016 年 1 月 15 日，William Morgan 和 Oliver Gould 在 GitHub 上发布了 Linkerd 0.0.7 版本。它用 Scala 实现，是业界第一个 Service Mesh 项目。2016 年下半年，Linkerd 陆续发布了 0.8 和 0.9 版本，宣布支持 HTTP/2 和 gRPC。2017 年 1 月 23 日，Linkerd 加入 CNCF。2017 年 4 月 25 日，Linkerd 发布了 1.0 版本。

Linkerd 在生产环境得到了大规模使用，但是随着 Istio 的诞生，Linkerd 开始走下坡路。由于 Istio 背后有强大的 Google 和 IBM 的支持，社区非常活跃，虽然到目前为止还没有大规模使用，但是业界已将 Istio 列为第二代 Service Mesh。2017 年 7 月 11 日，Linkerd 迫于压力发布版本 1.1.1，宣布和 Istio 项目集成，但是 Linkerd 在 Istio 中替代 Envoy 的难度还非常大，前景并不是特别乐观。



## Envoy

2016 年 9 月 13 日，Lyft 的 Matt Klein 宣布 Envoy 在 GitHub 开源，发布 1.0.0 版本。Envoy 用 C++ 实现，在性能和资源消耗上表现得非常出色。和 Linkerd 相比，Envoy 发展得更平稳，被 Istio 收编之后，Envoy 专注于数据平面。2017 年 9 月 14 日，Envoy 加入 CNCF，成为 CNCF 继 Linkerd 之后的第二个 Service Mesh 项目。目前，Envoy 已用于生产系统中，据 Lyft 介绍：“Envoy 在 Lyft 上可以管理一百多个服务，跨域上万台虚拟机，每秒可处理近两百万次请求。”

## nginMesh

2017 年 9 月，在波特兰举行的 nginx.conf 大会上，Nginx 在 GitHub 上发布了 nginMesh 0.1.6 版本。nginMesh 的定位是 Istio 的服务代理，也就是专注于数据平面，替代 Envoy，和 Linkerd 与 Istio 集成的思路一致。nginMesh 的发展一直比较缓慢，目前它还没有应用到生产环境中。

## Conduit

由于 Buoyant 公司在 Linkerd 上受到 Istio 的压制，2017 年 12 月 5 日，Buoyant 公司在 KubeConf 上发布了 Conduit 0.1.0 版本，作为 Istio 的竞争产品。Conduit 的架构几乎和 Istio 一样，也分成了数据平面和控制平面，为了性能和资源占用方面的优势，它直接采用 Rust 语言开发。Conduit 也被称为第二代 Service Mesh。

## Istio

2017 年 5 月 24 日，Istio 0.1 版本发布，目前已经到了 0.7 版本。Istio 背后是强大的 Google 和 IBM，所以 Istio 自诞生之日起就备受瞩目。Istio 的初期版本只支持 Kubernetes 平台，从 0.3 版开始支持非 Kubernetes 平台，并可以独立运行。

虽然它的架构思想被很多人认同，功能也很多，但是 Istio 的问题仍然较多，可用性较差，几乎没有生产级的案例。尽管如此，Istio 被认为是 Service Mesh 的未来，很多公司都在跟随这个框架。

实际上，如果在一个小公司，Service Mesh 的优势并不是十分明显。例如小公司很少会考虑采用多语言，因为多一种语言就意味着要花费额外的精力进行维护，所以到目前为止，大多数公司在后端只使用一两种语言。另外，微服务框架相对来说比较稳定，就如同 Spring Framework 一样，不会轻易进行不兼容的升级，只要保持好节奏，问题不会太多。

因此，像 Netflix、阿里这样的公司，基本上还是沿用类似 Spring Cloud 模式的。

此外，Service Mesh 在降低开发人员的使用门槛上也并没有那么明显，因为完全不考虑是不可能的，Service Mesh 在没有理解业务的情况下，是不能私自做决定的。例如 A 服务调用 B 服务的超时时间，设置多少合理呢？这需要开发人员根据业务配置。

另外，Service Mesh 对业务开发人员友好，但是基础设施层的研发会更复杂，需要依赖更多的服务、组件，否则将带来极大的架构、运维复杂度。对于很多公司来说，门槛稍高，且需要投入成本。

因此，对于绝大多数公司来说，Service Mesh 并不具备压倒性的优势，在没有成熟起来以前，大多持观望态度。

## 8.2.4 Istio 架构分析

Istio 架构如图 8-4 所示。Istio 从逻辑上可以分为数据平面（Data Plane）和控制平面（Control Plane）。如果用服务化框架类比 Istio，那么控制平面可以认为是注册中心及管理配置面板，数据平面可以认为是由服务化框架依赖的组件独立而成的一个进程，数据平面代理业务服务的注册发现、负载均衡、容错等能力。

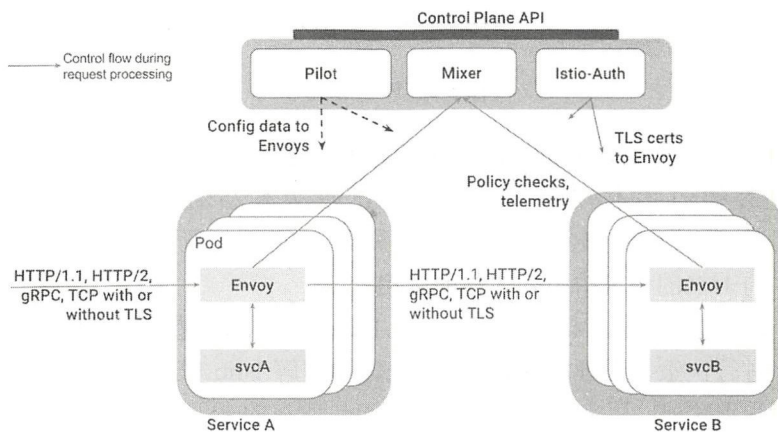


图 8-4 Istio 架构<sup>①</sup>

数据平面也就是代理，使用 Envoy 实现，独立进程和业务服务部署在一起，主要是管理微服务之间的网络通信及执行控制平面下发的指令。Envoy 支持的通信协议包括：HTTP/1.1、HTTP/2、gRPC、TCP 等，它支持 TLS。

<sup>①</sup> 图片来自 <https://istio.io/docs/concepts/what-is-istio/overview/>。

控制平面负责管理和配置这些代理，并动态执行策略。它主要包括三个部分，分别是 Pilot、Mixer、Istio-Auth，它们都是由 Go 语言开发的。

Pilot 主要的责任就是管理 Envoy 实例的生命周期。Pilot 通过 Envoy API 接入 Envoy 实例，主要用于服务发现及流量控制。运维人员可以通过 Rules API 自定义流量管理规则，转化为配置信息下发到 Envoy，Envoy 实例按照自定义的规则执行流控。从理论上说，Pilot 抽象了模型，通过适配器可以适配 Kubernetes、Mesos、CloudFoundry 等平台，但是目前只支持 Kubernetes。Pilot 架构如图 8-5 所示。

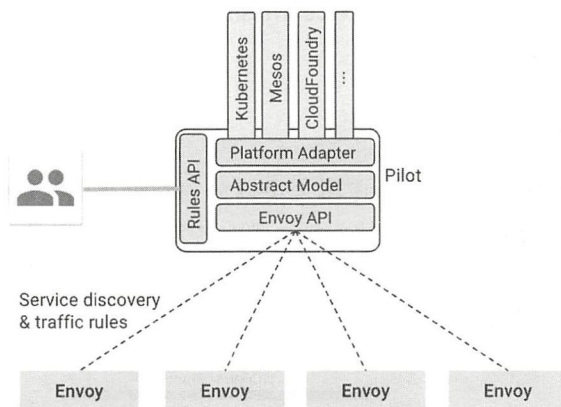


图 8-5 Pilot 架构<sup>①</sup>

利用 Pilot 可以轻松实现流量控制，如限流、灰度发布、熔断等能力。图 8-6 中的配置代表了 Pilot 分发到 v1 版本 75% 的流量，分发到 v2 版本 25% 的流量。

此外，利用 Pilot 还可以轻松注入故障。图 8-7 中的示例就表示在选择“reviews”服务的“v1”版本的 10% 的请求中引入 5 秒的延迟。

Mixer 负责通过策略进行访问控制，并从 Envoy 代理收集数据，代理提取请求级属性，发送到 Mixer 进行决策。要理解 Mixer，首先需要理解 Istio 中属性的意义。属性是具有名称和类型的元数据片段，用来描述流量和这些流量所在的上下文环境。请求的路径、大小、时间、源 IP 地址及目标服务如下所示。

```
request.path: blog/article
request.size: 200
request.time: 13:34:56.789 04/17/2018
source.ip: 10.10.0.1
target.service: article
```

<sup>①</sup> 图片来自 <https://istio.io/docs/concepts/traffic-management/pilot/>。



```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: reviews-v2-rollout
spec:
  destination:
    name: reviews
  route:
    - labels:
        version: v2
      weight: 25
    - labels:
        version: v1
      weight: 75
```

图 8-6 流量分发

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: ratings-delay
spec:
  destination:
    name: reviews
  route:
    - labels:
        version: v1
      httpFault:
        delay:
          percent: 10
          fixedDelay: 5s
```

图 8-7 示例

Mixer 提供如下三个核心功能。

- 校验。在响应之前，先验证调用者的身份是否是服务的白名单、是否通过 ACL 检查等。
- 配额。例如访问频率控制。当服务消费者对有限的资源进行争抢时，配额就成了一种有效的资源管理工具，它为服务之间的资源抢占提供相对的公平。
- 报告。使服务通过 Envoy 上报日志和监控。

Istio Auth 的目标是在不修改业务服务代码的前提下，提升业务服务之间通信的安全性。Istio Auth 利用 secret volume mount 从 Istio CA 向 Kubernetes 容器传递 keys/certs。由于 Envoy 代理一切流量，因此在 Envoy 客户端和 Envoy 服务端之间使用 TLS 非常简单。Istio Auth 为我们提供的能力如下。

- 每个服务都可以被设定身份，代表其角色，以实现跨集群和云的互通性。
- 可以加密服务间通信和终端用户到服务的通信。
- 提供密钥管理系统来自动执行密钥和证书的生成、分发、轮换和撤销。

在很多场景下，微服务通常不对外网开放访问权限，只有边缘服务才会通过 API Gateway 进行开放，而加密往往意味着降低性能，因此通常会选择不加密，除非部署的环境比较特殊。

# 9

## 第 9 章 研发流程

研发流程是一种组织方式，也是一种协议，研发流程的好坏直接决定产品的成败。亚马逊每年可以发布几百万次，规模小一点的互联网公司一年也发布几十万次，而传统型企业几年才发布一个版本，这完全不在一个层次上。

很多公司重视架构，而忽略了研发流程，研发流程能够决定研发效率，影响可用性、故障率等指标。在 Cloud Native 场景中，流程强调的是易用、标准、可靠，研发人员尽量通过流程工具达到自服务的效果，弱化约束、控制带来的影响。在软件的整个生命周期，应该如何开始，如何演进，如何结束？下面我们通过几个最佳实践来展开。

### 9.1 十二因子

十二因子<sup>①</sup>是 Heroku 开发团队在收集并观察了大量的 SaaS（Software as a Service）应用程序开发和执行过程后，整理出来的十二条开发 SaaS 应用程序的方针。它因被 Matt Stine 的 *Migrating to Cloud-Native Application Architecture* 一书引用，而为大众所熟知。十二因子能够帮助我们将传统架构迁移到 Cloud Native 架构，下面我们来分析一下。

---

①. 内容参考 12 factor 官网 <https://12factor.net/>。

## I. 基准代码。

一份基准代码，多份部署。

分析：使用 GIT 或者 SVN 管理代码，并且有明确的版本信息。代码仓库不只是开发人员提交使用，CI 也会每天执行下载代码、构建、打包、测试、运行等命令。另外，镜像（包含配置文件）、依赖的组件也应该统一管理起来。如果存在多个服务共享一部分公共代码的情况，应该将公共代码打包成类库（如 Java 中的 JAR）提交到依赖管理中进行共享，但是同一份代码可以部署多个实例。

## II. 依赖。

显式声明依赖关系。

分析：如果存在依赖，需要显示声明，否则在运行态，很有可能因为没有安装相关依赖导致失败。如 Java 中我们可以使用 Maven 或者 Gradle 管理依赖包，在工程配置文件中显式声明所有的依赖，即便开发人员分别使用不同的工具、不同的环境，也可以直接通过一条指令下载所有的依赖包。

## III. 配置。

在环境中存储配置。

分析：传统的管理方式基本上是运维人员手动修改配置文件，没有统一管理，一旦出现问题很难恢复。十二因子强调将代码和配置分离，因为在不同的环境中，代码是一致的，配置文件却大相径庭。这里需要说明的是，这里的配置不包括内部配置，强调的是类似数据库的连接地址一样的配置，而不是像 Spring 的 Bean 配置文件这种在任何环境都一样的配置文件。有几种方式可以解决这个问题，第一，可以将应用的配置存储于环境变量中；第二，可以将应用的配置存储于分布式配置中心。

## IV. 后端服务。

把后端服务当作附加资源。

分析：后端服务是指程序运行所需要的通过网络调用的各种服务，如数据库（MySQL、CouchDB）、消息/队列系统（RabbitMQ、Beanstalkd）、SMTP 邮件发送服务（Postfix），以及缓存系统（Memcached）。我们不用区别对待本地服务和第三方服务，所有的服务都作为后端服务去运行。后端服务强调的是作为服务，而不是中间件，要有独立的团队负责后端服务的整个生命周期。后端服务会自己维护、自己进化，使用者只需要通过接口调用，无须关注细节。



## V. 构建、发布、运行。

严格分离构建和运行。

分析：以前常见的一个典型错误是：代码已经打包好，在生产环境运行的时候才发现有 bug，开发人员经过调试后，发现可能是某个配置错了，或者是某个 html 引用的 JS 路径错误。为了简化，直接在生产环境修改代码。这是一个非常严重的问题，因为这些修改很可能没有同步到现有代码中，一旦回滚，可能会导致问题再次发生。十二因子强调通过 CI/CD（持续集成/持续布置）工具实现整个过程，例如使用 Jenkins。

## VI. 进程。

以一个或多个无状态进程运行应用。

分析：以多进程运行，首先要满足的就是应用无状态。如果存在状态，应该将状态外置到后端服务中，例如数据库、缓存等。当然，要尽量做到无状态，这样可以简化架构，但是并不是所有的应用都能保证无状态。例如长连接也算是一种状态，当客户端和某个服务端建立长连接的时候，从服务端发送消息到客户端，就需要找到是哪个服务端与客户端建立了连接。

## VII. 端口绑定。

通过端口绑定提供服务。

分析：应用通过端口绑定来提供服务，并监听发送至该端口的请求。服务绑定 IP 和端口，也意味着访问都是通过接口进行的。

## VIII. 并发。

通过进程模型进行扩展。

分析：扩展方式有进程和线程两种。进程的方式使扩展性更好，架构更简单，隔离性更好。线程扩展使编程更复杂，但是更节省资源。

## IX. 易处理。

快速启动和优雅终止可最大化健壮性。

分析：只有满足快速启动和优雅终止，才能使服务更健壮。如果一年要发布很多次，且并发请求又很多，那么当出现故障的时候能快速回退是很重要的。

## X. 开发环境与线上环境等价。

尽可能保持开发、预发布、线上环境相同。

分析：当开发人员把调试过的程序部署到测试环境或者生产环境时，往往运行不起来，

一个原因可能就是环境不同导致的。环境的不同会导致大量的故障产生，以前要保证环境等价是比较难的，而容器的出现使这件事做起来容易多了。

### XI. 日志。

把日志当作事件流。

分析：微服务架构中服务数量的爆发需要我们提供调用链分析能力，快速定位故障。除此之外，还需要将所有日志的输出规范统一收集到日志服务器中存储。这样开发人员能够轻易访问到日志，以便快速分析问题，解决问题。

### XII. 管理进程。

把后台管理任务当作一次性进程运行。

分析：一些工具类在生产环境上的操作可能是一次性的，因此最好把它们放在生产环境中执行，而不是本地，并且需要把它们像管理应用一样管理起来。例如定时任务，可以通过分布式任务调度平台统一管理。

## 9.2 为什么选择 DevOps

DevOps 的性能变化情况在 2016 年和 2017 年的对比，如表 9-1 所示。

表 9-1 2016 年和 2017 年的 DevOps 对比<sup>①</sup>

	2016	2017
代码部署频率	200 倍	46 倍
代码提交至代码部署实施速度	2555 倍	440 倍
停机后平均恢复速度	24 倍	96 倍
变更故障率	1/3	1/5

从表 9-1 中我们可以清楚地发现，高成效 DevOps 团队拥有如下表现。

- 高成效 DevOps 团队的代码部署频率是低成效团队的 46 倍。
- 高成效 DevOps 团队的代码提交至代码部署实施速度是低成效团队的 440 倍。
- 高成效 DevOps 团队的停机后平均恢复速度是低成效团队的 96 倍。
- 高成效 DevOps 团队变更故障率仅为低成效团队的五分之一（即二者变更故障比率为 1 比 5）。

交付速度、更新频率是衡量一个公司能力的重要指标。那些需要几年交付一次的软件

<sup>①</sup> 来自 2017 State of DevOps Report。

企业如果一旦面临以周或天为单位交付的软件企业的挑战，很可能被淘汰。一个典型的案例就是国内某手机制造厂商，在手机设计规划阶段，他们的手机是非常有市场竞争力的，非常超前的，但是一年以后手机才上市，早已失去了优势。因为热点已经发生转变，竞争对手在相应时间内已经发布了两三个版本。

敏捷开发流程帮助我们实现了快速反馈、快速验证，交付到生产环境是一个非常烦琐的过程，实际的表现就是开发人员熬夜变更，在流量重新恢复之前完成。在此过程中开发人员要抗住各种压力，聚精会神，不能写错一行命令。结果可能是不尽如人意，各种故障层出不穷。开发人员怪运维人员写错命令，运维人员怪开发人员留了好多“坑”，出故障了怪测试人员测试不充分，测试人员怪开发人员“留坑”太多，时间太紧没法全找出。

我们可以看一下 Amazon 这么多年来一直坚持 DevOps 的成果，2011 年平均每天部署 7 000 次，2015 年的这个数字是 13 万次，2006 年到 2011 年部署引发的停机减少 75%。这是如何实现的呢？如果每次变更都要耗费如此巨大的精力，是不可能完成的。下面我们为大家详细介绍一下 DevOps 模型。

Gartner 提出的 DevOps 模型，它分为四个部分，包括文化、技术、过程和人。而很多团队失败或者无法充分发挥 DevOps 优势的原因往往是由于人和文化，人的思维和团队的文化才是最难改变的。Gartner DevOps 模型，如图 9-1 所示。

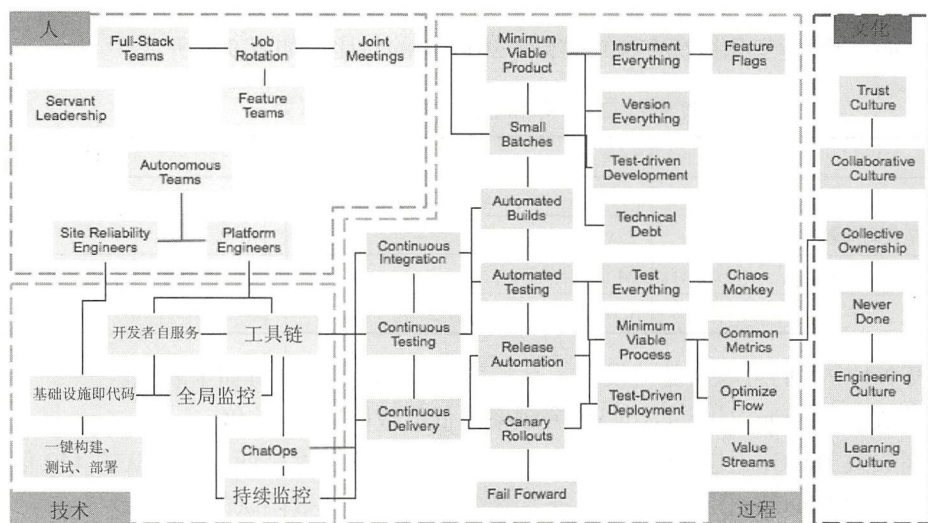


图 9-1 Gartner DevOps 模型<sup>①</sup>

Gartner DevOps 模型的技术方面包括，基础设施即代码、全局监控、持续监控，我们

<sup>①</sup> 图片来自 <https://www.gartner.com/en>。



在基础设施章节已经做了具体的描述，下面给大家介绍工具链、一键构建、测试、部署、开发者自服务。

## 9.3 自动化测试

自动化测试是一个比较宽泛的概念，理论上讲，所有代替人工测试的方法都可以认为是自动化测试。随着 DevOps 的盛行，为了减少沟通，测试工程师可能会随着全栈工程师的到来而变得越来越少，现在很多公司已经开始不招测试工程师岗位了。那么测试的工作应该谁做呢？测试成了全栈工程师的工作，因为不沟通才是最有效率的沟通。

如果要求你按天进行交付，那么传统的开发流程、测试方法是行不通的，人工测试根本就来不及。传统的开发流程更像一个流水线，每个人负责一个环节，经常出现开发人员提交的代码为不可测试状态，部署到测试环境甚至都运行不起来等状况。

### 9.3.1 单元测试

我们希望代码能做到什么？

- 满足功能需求。
- 质量好，没有 bug。
- 可维护，很容易读懂、修改。

而单元测试就是为了实现后两个目标的，提升代码质量和可维护性。实际上，很多公司并不注重单元测试，要么公司对测试覆盖率有要求，后来去补；要么根本不写，凭借测试人员人工测试。

开发人员普遍反馈的问题如下。

- 测试代码太多，严重浪费时间。
- 实现单元测试太复杂，例如多线程。
- 有专门的测试人员负责，开发人员应该做更有价值的工作。
- 以前没有写单元测试，结果也没啥问题。

如果实现一个非常简单的功能，假设在 1 000 行代码以内，可能不会有太复杂的逻辑，凭借优秀的代码能力，完全可以使程序在没有单元测试的情况下正常运行。但是当代码达到一定规模的时候，你的逻辑可能会出现混乱，或者过了很长一段时间，你的记忆变得模糊，根本记不清楚当初是怎么考虑的，改哪个参数会影响什么地方，这时可维护性受到影响，导致代码积重难返，怎么办呢？不动以前的，增加一个实现，让测试人员全部测试一遍，要知道，这可能会花费很长时间，而且若测试不充分，这就很可能导致质量问题。

如果此时要求我们持续交付，假设单元测试执行一次需要两分钟，人工测试一遍需要 1 人花费 1 天时间，则每周甚至每天都上线一次，单元测试会节省多少时间和人力成本呢？这时候人工测试就会显得非常吃力了，并且容易出现问题。

另外，当你去实现单元测试代码的时候，实际上是对代码结构的一种检测，当代码结构不合理的时候，特别是一个大函数，几千行代码，单元测试写起来会相当困难，单元测试实现过程可以让你重新审视代码结构，强制你实现结构化更好、逻辑性更强的代码。

但是，当代码量很少，逻辑很简单，开发人员也非常熟悉代码的时候，单元测试确实显得很多余，随便点几个按钮就能解决问题。但是，往往现实的情况是，需要很多开发人员一起协作开发，代码逻辑混在一起，开发人员更换速度很快。此时，单元测试就显得尤为重要。

单元测试也不必覆盖所有方法，因为有一些已经用过很多次的方法，或者一些非常简单的、自动生成的代码，再去单纯为了提升覆盖率而实现单元测试就没有必要了。单元测试应该首先覆盖到边界，边界是容易出错的地方，还有复杂逻辑的方法及修复的 bug。

### 9.3.2 TDD

TDD（Test-Driven Development）即测试驱动开发，是一种设计方法，而不仅仅是测试方法，它基于极限编程（XP）中测试优先的理念，是一种测试代码先于编写代码的思想，要求在代码实现前考虑实现代码需要提供的功能。TDD 的原理是在开发功能代码之前，先编写单元测试用例代码，用测试代码指导开发者编写产品代码。

TDD 的流程大致如下<sup>①</sup>。

- (1) 编写一个测试代码。
- (2) 运行所有测试代码——红灯。
- (3) 编写实现代码。
- (4) 运行所有测试代码——绿灯。
- (5) 重构。
- (6) 运行所有测试代码——绿灯。

在编写实现代码的时候，不要试图让实现代码完美无缺，每次应该只关注一项功能，实现代码只是针对刚刚编写的测试代码的，剩余的实现代码应该由另外的测试代码驱动。

整个流程强调快速在测试和实现之间来回切换。严格遵循“红灯-绿灯-重构”的步骤，是一个从失败到成功再到完美的过程。也就是说，在编写完一个测试的时候应该是红灯，

---

<sup>①</sup> 观点参考 <https://www.railstutorial.org/>。

测试失败，否则一定是测试代码有问题。编写代码实现之后，运行所有测试时，应该处于绿灯状态，全部测试通过。如果不能通过，则说明实现代码有问题，必须修改实现代码。当全部测试通过后，处于绿灯状态时，应该考虑一下是否需要进行重构，最终全部测试通过，处于绿灯状态。

### 9.3.3 提交即意味着可测试

提交后测试失败通常由以下几个原因引起。

- 代码编译错误。
- 代码逻辑错误。
- 环境不一致。

传统的测试需要设置固定时间点，当到达某个时间点才能测试，而 DevOps 强调的是尽早识别错误，错误越早被发现就越容易被修复，因为提交代码的人对上下文印象深刻，发现和解决问题的速度会比几天后快得多。因此测试没有固定时间点，只要你提交了代码，就意味着进入了测试阶段。因为代码一旦提交，持续集成工具会定时发布到指定的环境中。如果某个功能没有完成，则应该使用特性开关将未完成的功能关闭。

在这种情况下，单元测试就显得尤为重要。如果采用 TDD 的方式，达到这个标准完全没有问题。

在代码提交前，应该更新所有服务端的代码并进行合并，运行测试，没有问题再进行提交。

## 9.4 Code Review

Code Review，顾名思义，就是针对一名开发人员完成的代码，让团队其他开发人员检查的过程。很多公司都在开展 Code Review，但是绝大多数公司只是流于形式，并没有形成一种文化。Code Review 更多依靠的是小团队的工匠精神和程序员个人的主观能动性。

### 9.4.1 Code Review 的意义

Capers Jones 对超过 12 000 个软件开发项目的持续分析表明，正式进行 Code Review 的潜在缺陷发现率在 60%~65%，对于非正式 Code Review 的情况，这个数字不到 50%，而不进行 Code Review，直接进行测试的潜在缺陷发现率约为 30%。一般的代码审查速率为每小时 150 行代码，代码审查最高可以达到 85% 的缺陷去除率，平均缺陷去除率约为 65%。

大多数拒绝 Code Review 的团队都会把原因归结为“没时间”，这的确是他们的真实



原因之一，而在大多数时候写代码不会像“搬砖”那么简单，“技术债”带来的后果是非常严重的。下面看一下 Code Review 能给我们带来什么。

- 发现潜在的 bug。可能有一些 bug 是隐藏的，测试人员或者自动化测试代码无法轻易发现，这种 bug 需要满足一些特定的条件，发生的概率比较低，但是一旦发生，其破坏性会是致命的。Code Review 可以发现一部分潜在的 bug。
- 提升代码易读性。如果将代码交给别人，并且能让别人看懂，那么就要求在命名、注释方面特别讲究，晦涩难懂的代码谁也不愿意看。
- 统一规范、标准。很多时候标准、规范是一纸空文，除了通过工具检查，一些潜在的风格，可以在 Code Review 的过程中实现。
- 技术交流，提升能力。Code Review 实际上也是学习其他人代码能力、架构思路的过程。新人可以在 Code Review 过程中得到锻炼。

Code Review 不仅仅是开发流程中的一个重要环节，更是一种公司开放、自由的文化。当这种文化建立起来的时候，Code Review 的实施不需要强制手段，不需要明确的惩罚措施。

## 9.4.2 Code Review 的原则

为了让 Code Review 可以长期运行，我们需要建立一套适合的体系。首先，我们要做的是统一思想，下面这些原则可以帮助我们做到这一点。

- 以发现问题为目标。团队需要开放、透明，整个 Code Review 的过程是对事不对人的。如果害怕得罪人，只是象征性地提几个不痛不痒的问题是没有意义的，或者认为这个问题一旦提出会导致大规模重构，干脆放弃。如果认为有问题，则一定要提出，至于什么时候解决，或者是否解决，那是另外一回事，真实的情况是，并不是所有的问题都能得到及时解决。
- 不设置惩罚。一旦涉及惩罚，可能会导致矛盾激化。如某团队推行的，在 Code Review 的过程中，每找到一个问题加一分，被找到一个问题减一分，以此作为年终绩效的一部分，有的人会花费大量时间去找一些模棱两可、不痛不痒的问题，导致团队矛盾重重。经验告诉我们，不应该设置惩罚，而应该采用正向激励的做法。
- 不论资历。不要认为资深的开发人员就没有问题，不会犯错误，世界上不存在没有 bug 的软件，问题少不代表没有问题，大家都是平等的，只看问题不看人。此外，在 Code Review 过程中仔细推敲优质代码，可以学习到很多知识。

### 9.4.3 Code Review 的过程

准备工作很重要。大多数人刚开始参加 Code Review 的时候都带着很多误解。我见过的很多 Code Review 的准备工作做的都不尽如人意。团队成员会担心“我的问题被发现会受到什么惩罚？”“会不会很丢人？”“扣不扣 KPI？”“会不会失去领导的信任？”……而检查别人代码的时候也会面临“如果我提出异议会不会导致别人面子上过不去？”“会不会导致我的人格魅力下降？”“老员工的代码一定不会有问题，有问题可能是我没有理解。”“一个问题都找不到，是不是说明我很无能？”等一系列的问题。所以在 Code Review 开始前，应该在团队内部充分沟通，描述 Code Review 的重要意义，特别应该强调它对个人成长和代码质量提升的重大意义。

线上 Review，利用工具，事先规定好 Review 的范围、任务，每个人抽出一定的时间进行 Review，发现的问题可以直接基于工具在代码上进行标记，也可以及时修复。线上 Review 的时间更灵活。但是通常刚刚接触 Code Review 的团队，在进行线上 Review 时效果并不是特别好，因为很多人会偷懒，或者不知从何处入手。当整个团队对 Code Review 比较熟悉了，Code Review 的团队文化建立起来了，采用线上 Review 的方式效果会更好。

线上 Review 的工具有几十种，例如依托 Eclipse 的 Jupiter、ReviewClipse。很多互联网公司也有自己的解决方案，开源的有 Facebook 使用的 Phabricator、阿里使用的 Tao-ReviewBoard。

线下 Review 通常是在一个会议室内集体进行检查。大家有充足的讨论时间，对思路、业务的理解更快，检查的人能集中精力，达到的效果通常要比线上 Review 好，但是线下 Review 需要耗费大量的人力，不宜频繁进行。

也可以将线上 Review 和线下 Review 结合起来用，隔一段时间进行一次线下 Review。

## 9.5 流水线

### 9.5.1 持续交付

让软件交付生命周期中的所有参与者，包括业务团队、架构师、开发人员、测试人员，以及 IT 运营和生产人员都围绕持续创新这一共同目标而努力并协调地工作。在持续交付的推动下，通过持续反馈的不断塑造，最终实现持续创新。持续交付能够给我们带来如下优势。

- 降低交付周期。
- 通过自动化工具实现设计、开发、测试、发布、运维各个阶段的重复性工作。
- 通过工具实现标准化、规范化，降低出错概率。

一般企业交付场景分为自运营场景和项目交付类，自运营场景完全可以做到持续交付，项目交付类可以先实现持续集成。

实现持续集成需要做到如下几点。

- 以代码为基准，代码统一管理。
- 单元测试覆盖率高。
- 自动化功能测试和集成测试。
- 代码提交意味着进入测试环节，持续集成工具每天定时从 SVN 或者 Git 中拿代码发布到测试环境，编译或部署失败时自动发布告警。
- 通过分布式配置中心实现功能开关，未完成的功能可以暂时关闭。
- 持续集成工具可以集成代码检测、测试覆盖率、代码重复率等代码质量监控。

持续交付在每个公司推行的最大困难都是对旧习惯的打破。

### 9.5.2 持续部署流水线

1910 年，福特汽车公司在引入生产流水线之后，Model-T 大组装时间缩短到原先的八分之一，从 12.5 小时降到了 1.5 小时，这就是流水线改变世界的的神话。

什么是软件部署流水线？

Jez Humble 在《持续交付》一书中对部署流水线的定义是：部署流水线是指软件从版本控制库到用户手中这一过程的自动化表现形式。软件的每次变更都会经历一个复杂的流程才能发布。

图 9-2 是 CNCF 关于持续交付的一些技术栈，常用的例如 Jenkins、Bamboo、Travis CI、Spinnaker、GitLab-Runner 等。

由于部署流水线和代码仓库、项目管理工具、基础设施都存在着强烈的依赖关系，而每个公司的发布流程又千差万别，结果导致大多数有实力或者重视交付能力的公司多采用自研的方式，但是流程都差不多。

### 9.5.3 基于开源打造流水线

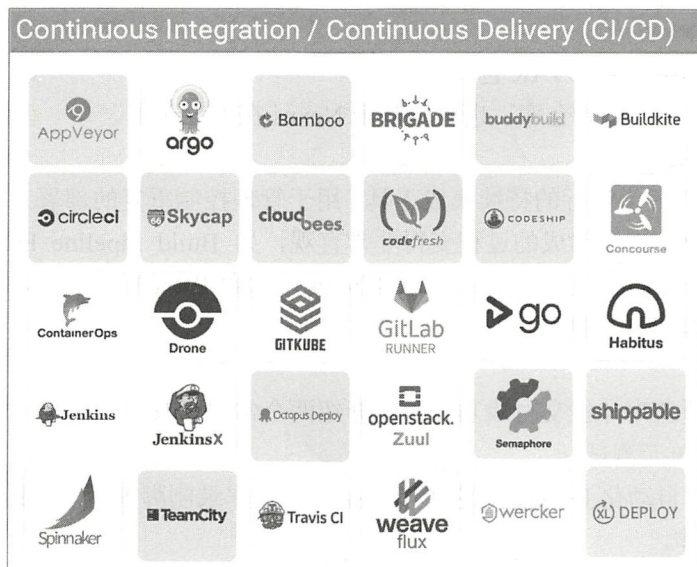
下面我们以 Java 环境为例，介绍一下常用的流水线工具。

#### (1) 需求管理。

需求管理的作用是管理需求，可以基于需求做项目跟踪管理。下面介绍两个常用工具。

- JIRA 是 Atlassian 公司出品的项目与事务跟踪工具，被广泛应用于缺陷跟踪、客户服务、需求收集、流程审批、任务跟踪、项目跟踪和敏捷管理等工作领域。JIRA 的优点是配置灵活、功能全面、部署简单、扩展丰富。JIRA 在国内使用广泛。



图 9-2 持续交付技术栈<sup>①</sup>

- Kanboard 是基于 PHP 开发的看板工具，它是 GitHub 开源的一款优秀的项目管理软件。它可以让你更有效、更简单地管理项目。

### (2) 代码仓库。

代码仓库的作用是统一代码仓库，以代码为标准。下面介绍两个常用工具。

- Git 是目前最流行的开源分布式版本控制系统。它有强大的分支管理能力，非常适合协作开发，是敏捷开发团队的最爱。
- SVN 是集中式的老牌版本管理系统。

### (3) 构建工具。

构建工具的作用是自动化构建，可以和 CI 工具集成使用。下面介绍两个常用工具。

- Maven 是当下最流行的构建工具，功能已经不仅仅是构建了，还包含了一个项目对象模型（Project Object Model）、一组标准集合、一个项目生命周期（Project Lifecycle）、一个依赖管理系统（Dependency Management System），和用来运行定义在生命周期阶段中插件目标的逻辑。
- Gradle 是一个基于 JVM 的构建工具，支持 Maven、Ivy 仓库，支持传递性依赖管理。它简化了 Maven 烦琐的配置文件，是一款灵活通用的构建工具。

<sup>①</sup> 图片来自 <https://www.cncf.io>。

#### (4) 共享仓库。

共享仓库的作用是共享 JAR 包。

Nexus 可以访问远程的仓库，也可以基于 Nexus 在本地实现。

#### (5) 持续集成。

Jenkins 是当下最为流行的持续集成工具，用于管理控制重复的部署工作。利用 Jenkins 的插件，可以让整个持续集成的过程变得非常直观，如 Build Pipeline Plugin。Kubernetes Plugin 可以和 Kubernetes 进行集成；JIRA Plugin 可以和 JIRA 集成起来，项目管理人员通过 JIRA 可以了解项目的进度。

#### (6) 测试。

更高的测试覆盖率才能交付成功。下面介绍两个常用工具。

- Junit 用于单元测试。
- Selenium，自动化 UI 测试，它可以模拟人的操作录制脚本，让工具每隔固定时间进行一次操作。

#### (7) 监控。

监控的目的是一旦出现问题，可以立即做出响应，包括告警、回滚。下面介绍几个常用工具。

- Nagios 是开源的免费网络监视工具，可以开发很多插件进行业务监控。例如对于数据库中某张表的增长速度的监控。
- Zabbix 是分布式系统监视及网络监视功能的企业级开源解决方案。
- Prometheus+Grafana，前面有介绍，此处不再赘述。

从理论上说，打造流水线的整个流程大致如下。

#### (1) 基于项目管理工具分析需求，编写测试用例。

(2) 基于流水线创建服务，在创建服务的时候可以直接生成 YAML 格式的描述文件，并生成相关代码提交到代码仓库，流水线创建开发、测试、预生产、生产环境。

(3) 开发人员到代码仓库下载代码，在本地可以直接运行。开发后提交代码到代码仓库。

(4) 流水线定时在测试环境进行打包、编译、构建、部署，并进行自动化测试。生成相应依赖性 JAR 包，发送失败通知和报表邮件。

(5) 开发完成后，一键发布到预生产环境中，进行相应测试，此时处于预上线状态。

(6) 灰度发布，上线。

当然，通过以上工具搭建的环境还不够自动化，工具之间缺乏整体的贯通性。下面我们参考一下 Amazon 的流水线。

### 9.5.4 Amazon 的流水线

Amazon 是业界公认的微服务架构的开创者，当他们采用微服务架构的时候，自然会遇到微服务架构带来的问题，因此 Amazon 的一个非常重要的“武器”就是全流程的工具链。全流程的工具链主要由以下几部分组成。

- Octane，可以通过界面自动生成前端页面和后端服务，自动生成 Brazil 包和 Apollo 环境，并且自动部署到 Pipeline 中。在 Amazon，微服务框架统一使用 Coral Service，在这里可以基于 Coral Service 生成代码。
- Brazil，以 Eclipse 插件的形式管理项目之间的依赖，提供统一的代码仓库，通过命令构建和部署服务。
- Apollo，部署和配置环境工具，自动化伸缩，全生命周期的监控、管理、配置服务。
- Pipeline，自动化持续集成和部署，发布权限控制，流程审批。

我们可以看到开发流程是整体贯通的。

### 9.5.5 开发人员自服务

在开始介绍开发人员自服务之前，先举一个生活中的例子。20 世纪 80 年代，我们去商店买一包糖，会有专门的服务员帮你拿过来；如果不是自己喜欢的那一种，则会让服务员拿回去，再拿另一种来，挑选到满意的以后再称重、结算。这个过程的效率实际上很低，人多的时候需要排起长队。后来开始有了超市，东西非常全，需要什么自己拿，大多数物品已经提前称重并标上价格了，超市的模式超越了以前的柜台模式，但是结算时仍然要排长队。如今，开始有了无人超市，要什么自己拿，拿完了自己结算，效率提升了很多，包括现在有的大超市开始有自助结算通道，连麦当劳都有了自助点餐台，这是不断追求效率和体验的结果。

对于开发过程来说，少交流、少沟通、少开会并且不会导致做错，就是最高的效率。而 DevOps 的环境，包括自动化的研发环境、周密的上线过程、智能化运维，为一切做好了铺垫。从需求到上线完全可以由一个人完成，除了前期和产品人员交流，无须频繁地对外沟通，甚至当生产环境出现故障的时候，在大多数情况下，也是开发人员一个人解决的，中间没有了测试人员和运维人员。要知道，在传统开发流程中，开发人员、测试人员和运维人员的沟通是非常频繁的，并且非常容易出错。

当然，这一切的基础至少包括如下几点。

- 高覆盖率的自动化测试。
- 全面的监控。
- 持续交付流水线。





- 敏捷基础设施。
- 自动化/智能化运维。
- 好的架构。
- 全栈工程师。
- 服务型管理。
- 工程师文化。
- 信任文化。
- 分享文化。

## 9.6 为什么需要 AIOps

AIOps 是 Gartner 在 2016 年提出的，即 Artificial Intelligence for IT Operations，译为智能运维。简单来讲，就是将人工智能应用于运维领域，基于已有的运维数据（日志、监控信息、应用信息等），不依赖于人工，通过机器学习的方式来解决自动化运维没办法解决的问题。Gartner 的报告宣称，到 2020 年，将近 50% 的企业将会在他们的业务和 IT 运维方面采用 AIOps，远远高于现在的 10%。

为什么要用 AIOps 呢？

- 大量的故障误报怎么办？

大多数运维人员及开发人员都经历过半夜被告警吵醒，登录到系统发现是一条误报信息，生产环境没有任何问题的状况。但是这类告警又不能关掉，宁可错杀一千，也不放过一个。这些告警有多大比例呢？根据我的统计，大概能占到四分之一，这将消耗巨大的人力。来自腾讯的一组数据，他们每天处理的告警数据大概有十几万条，而通过 AIOps 处理在 2015 上半年节省人力 10 214 小时（预警不计算人力节省），保守估计相当于 7 个运维人员每天 8 小时三班倒值守的工作时长。

- 故障如何自动恢复？

当系统到达一定规模的时候，任何时刻的中断都会导致大量的金钱损失和用户损失，因此对系统可用性的要求也就越来越高。如果按照 5 个 9（99.999%）计算，那么每年系统不可用的时间只有 5 分钟，人工解决问题肯定是来不及的，只能想办法让服务自动恢复。

《企业级 AIOps 实施建议白皮书》<sup>①</sup>将 AIOps 的能力分为如下五级。

- 第一级——开始尝试应用 AI 能力，还无较成熟的单点应用。

---

<sup>①</sup> 由高效运维社区和云计算开源产业联盟（OSCAR 联盟）牵头，互联网企业如百度、阿里巴巴、腾讯、360、华为、平安科技等公司的 AIOps 负责人联合编写了国内外首本《企业级 AIOps 实施建议白皮书》。





- 第二级——具备单场景的 AI 运维能力，可以初步形成供内部使用的学件。
- 第三级——有由多个单场景 AI 运维模块串联起来的流程化 AI 运维能力，可以对外提供可靠的运维 AI 学件。
- 第四级——主要运维场景均已实现流程化免干预 AI 运维能力，可以对外提供可靠的 AIOps 服务。
- 第五级——有核心中枢 AI，可以在成本、质量、效率间从容调整，达到业务不同生命周期对三个方面不同的指标要求，可实现多目标下的最优或按需最优。

举一个简单的例子来说明 AIOps 的作用：某登录功能要设置超时告警，在没有使用 AIOps 之前，一般会设置一个固定值，如 1 秒，超过则发送告警。但是，系统并不是平稳运行的，有的时候用户多，有的时候用户少，在用户多的时候，超时告警时间设置为 1.2 秒可能比较正常，而当用户少的时候，超时告警时间设置为 0.5 秒就不太正常了。因此，采用一个静态值是比较麻烦的。

如果我们采用 AIOps，那么如何解决这个问题呢？

收集近期的（如一年或半年的数据，数据越多，准确性越高）登录时长数据，通过 Tensorflow 进行训练，训练完成后，Tensorflow 可以给出预测值，把现有登录数据和预测值进行比对，偏离过高则发出告警，进行弹性伸缩。

## 9.7 基于数据和反馈持续改进

从产品的角度，一味地叠加功能是绝对错误的。大家从微信就可以看出来，用户需要的往往是有限的几个功能，那些复杂的、炫酷的功能堆积多了反而会导致用户体验急剧下降。否则，还能指望老人会用微信吗？我相信，如果微信交给某传统企业开发，最终会变成一个无所不能的巨无霸，而另外一个简洁的产品可能会迅速取代它。

那么，一个产品经理做了一个功能，怎么知道是好还是不好呢？对了，我们需要数据分析反馈和用户反馈。敏捷开发强调持续反馈能力。反馈越早，效果越好，成本越低。

- 用户行为分析。基于大数据分析，从用户访问的日志中寻找规律，统计发现行为数据。
- 用户反馈。反馈功能在每个互联网产品中都必须有，而且用户反馈的内容并不少，包括用户使用产品的感受，要求新增某个功能，甚至使用过程中的疑问、故障。每个产品经理都应该重视用户反馈，并及时回复。
- A/B 测试。最早可以是公司内部使用，然后选取一批友好用户体验并进行反馈，根据反馈结果进行优化，整个过程中，出现任何问题应该及时变更。





## 9.8 拥抱变化

一个案例，以前产品总监问我：“你们为什么总是延期？”

我反问：“那你能不改需求吗？”

他说：“不能。”

显然，我们都意识到不改变是不可能的，那就要拥抱变化。张小龙曾经说过，“他们上线前两小时是允许变更需求的。”

那么怎么才能拥抱变化呢？

首先，尽早变更。产品先以一个可视的原型展现，越早发现问题，做出修改的代价就越小。

其次，缩小决策范围。我们需要一个快速有效的决策机制。决策的人级别越高，效率就越低。但是如果决策的人级别太低，又可能造成大量的意见，不能服众。最好的方式是让那些有决策能力的、对产品特别熟悉的人中最低级别的人来做决策。

再次，设置优先级。所有的任务都应该设置优先级，当做出承诺的时候，只承诺高优先级的时间计划。

最后，一切都自动化。要做好自动化测试、一键发布、灰度发布、故障快速定位跟踪、故障快速恢复等措施，以应对频繁变更。

## 9.9 代码即设计

计划软件设计强调架构师提前做好规划，提前把问题考虑周全，他们可能没有更多时间去写代码，所以采用 UML 等方式摆脱程序设计的细节，一旦设计完成，完整的交给另外一个开发组织实现。老板们认为架构师比较昂贵，用他们写代码简直就是浪费人才，可能你已经看出如下几个问题。

- 架构师的想法未必能完整传递给开发人员，为什么要建索引？为什么要冗余？你不可能要求架构师把所有问题都覆盖到。技术实现上的任何一个失败点都可能导致对外表现的巨大差距。
- 大量的精力都耗费在设计文档之上，一旦中间有任何需求或者技术上的改变，可能的结果是研发人员直接实现，而前期耗费巨大精力的设计文档不能得到同步更新。
- 架构师久而久之失去了技术敏感度，不只是在技术能力上，有可能导致研发人员失去了信心，得不到尊重，进而导致内部不和谐。

这就是 XP 所提倡的代码即设计。XP 中经典的两句话是 “Do the Simplest Thing that







Could Possibly Work”和“You Aren't Going to Need It”。后者就是最著名的“YAGNI 理论”，它表示你不应该为明天可能会用到的代码，把大部分开发精力都浪费在今天。否则你需要支付很多额外的费用，而明天可能会有变化。当然，并不是说不需要设计，而是要把设计的过程分配到整个开发过程中。传统研发流程与敏捷研发流程的对比，如图 9-3 所示。传统研发流程强调阶段性，而敏捷研发流程的阶段性更模糊。

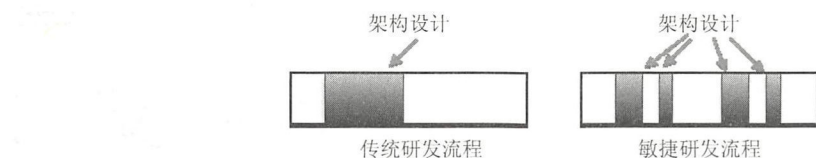


图 9-3 传统、敏捷研发流程对比

敏捷研发流程模糊阶段性的理由是：业务需求太多和技术变化速度太快。

架构设计包含两方面，一是架构，二是设计。在敏捷开发中，架构并没有被抛弃，架构的思考可能发生在理解需求的过程中，也可能来自你的经验；架构的结果可能是一个白板上简单的图，也可能需要详细调研，这与架构师的能力有关。设计需要耗费大量的时间和精力去做，设计会转移、分配到整个研发流程。

这和 Martin Fowler 所提倡的重构有异曲同工之妙。整个进化设计需要简单的架构+持续集成+重构+整个研发流程的设计来保证。

另外一种观点认为前期设计还是比较重要的，它为软件打好基础，避免不必要的重构，文档是传递的基础，有必要花精力去维护文档。我的观点是设计应该尽可能保持简单，不做过度设计，我们不可能在前期设计出面面俱到的软件，需要依赖后期的实现。

我们要根据需求变化的速度、团队的技术水平、是否可以做到持续交付确定架构设计的流程。前期的不确定无疑需要我们保持清晰、简单的设计，团队的技术水平越高，我们的架构设计就可以越简单。如果我们选择一个所有人都不熟悉的框架，那么需要额外做很多技术预演。另外，这里的架构是基于云的架构，很多交付型的软件不能做到持续交付，只能在前期花更多的时间来设计，但是我并不提倡耗费大量的精力做详尽的文档。最好是团队一起来做架构设计，架构师起引导作用，在设计完成后和开发人员一起来实现。我相信团队的力量大于个人的力量。无论如何，保持进化的设计是非常有诱惑力的。

这种设计上的转变实际上非常适合小规模、有强大战斗力的团队。因为开发人员以前只是编码，现在需要设计能力了。每个人都会感觉自己更有价值，得到了能力上的提升，有利于培养新人。整个团队的沟通情况也更加顺畅，就算出现人员更替，也能更从容地应对了。这实际上跟微服务强调的研发流程及文化是一脉相承的。





# 10

## 第 10 章 团队文化

如果你穿着西装，打着领带去参加一个重要的会议，那么你绝对不会在会场内随便扔垃圾，这是环境给你的暗示，跟制度、规范无关。好的文化会帮助团队营造一种舒适的环境，让每个成员工作更加愉悦。正所谓，强势文化造就强者，弱势文化造就弱者。在体育界，有一个词汇叫“冠军文化”，这就是一种团队文化，每个团队都有自己的性格，当遭受挫折的时候，是一蹶不振，还是强势反弹？冠军文化可以让一支球队在最关键的时刻咬牙挺住，而挺住就意味着一切。

团队文化就好比土壤，要培养什么样的员工，就要有适合他的土壤。传统企业常常去互联网公司挖一些高级技术人才，但是挖过来之后，高级技术人才要么很快就辞职了，要么失去了以前在互联网公司的风采，一个很重要的原因就是无法适应传统企业的团队文化。好的文化会吸引好的人才，并激励他们留下来用最佳的状态工作。

### 10.1 为什么团队文化如此重要

一个经验丰富的职场人士，通过日常工作就能感受到一个企业的文化是开放的还是封闭的，是自由的还是严谨的，是更重视长期发展还是更重视目前收入。管理大师德鲁克曾经说过：“Culture eats strategy for breakfast”。大概意思是说“文化可以把战略当早餐吃掉”，在不好的文化面前，再好的战略也会在一顿早餐后忘掉，团队文化才是战略中的“战略”。





每个伟大的公司都有自己非常清晰的团队文化。如 Facebook 强调的是分享、连接, Airbnb 强调的是归属感, Google 强调的是透明、发声的权利。

在传统的研发模式下, 我们往往忽略了团队文化的重要性, 研发流程更像流水线, 而研发人员则像流水线上的工人。实际上, 一个效率很高的工程师完成的工作量可能是一个效率低下的工程师的几十倍甚至几百倍。

传统企业和互联网公司所信奉的管理文化差异非常大, 下面通过一个例子来估算一下成本。

在传统企业, 技术经理接到一个单点登录的功能需求, 架构师 A 大概花了一个星期调研, 又花了一个星期编写 PPT, 在某个内部会议上进行架构决策, 会议中各方面领导提出了各种质疑, 认为某开源方案缺陷太多, 漏洞百出。会后 A 进行返工, 最终一个月时间才确定了架构文档, 而实际负责开发的可能是工程师 B、C、D。A 分别对 B、C、D 进行了大量的解释, B、C、D 又给测试人员、运维人员进行了大量的解释。最终这个功能大概花了一个架构师和三个工程师各 2 个月时间, 一个测试人员 1 个月时间, 以及一个运维人员半个月时间。如果架构师的薪资为每月 5 万元, 开发人员每月为 2 万元, 测试人员每月 1 万元, 运维人员每月 2 万元, 那么这个功能预计要花费 24 万元才能实现。

在互联网公司, 根据前面的任务拆分, 某工程师 A 领到了单点登录的功能需求, 花了一个星期进行调研, 认为某开源产品可以满足 80% 的需求, 经过团队内非正式的讨论, 最终决定基于开源产品进行改造。A 花了一个星期研究源代码, 一个星期进行测试, 再经过一轮内部讨论, 然后花一个月时间改造代码, 一个月时间测试及上线。整个过程不到三个月时间都是以 A 为主导, 如果这个工程师的能力比较强, 薪资大概是 5 万元, 和传统企业的架构师持平, 那么这个功能预计要花费 15 万元。

当然, 这只是一个粗略的计算, 有人可能会质疑, 互联网公司打造研发环境、工具所消耗的费用, 同样互联网公司也没有计算技术经理/项目经理的管理费用。一个实际的例子, Instagram Video(15 秒短视频)从需求到上线花了 30 个人一个月的时间, 包括 iOS 和 Android 客户端、服务端, 甚至包括市场及推广, 在这期间为了解决防抖的问题, 甚至花了 7 天时间收购了一家公司集成进来<sup>①</sup>。

李开复曾经说过: “我们进入了信息社会, 这个社会跟工业社会不一样的就是, 顶尖人才和普通人才的差异化不再是 20%、30% 了, 而是五倍、十倍甚至一百倍的差距。”

从结果上来讲, 互联网公司的研发模式效率要更高, 除了流程的问题, 还有一个重要

<sup>①</sup> 来自 Instagram 工程师陈晨的介绍, 地址 [http://www.infoq.com/cn/presentations/the-culture-and-management-of-western-engineers?utm\\_source=infoq&utm\\_medium=videos\\_homepage&utm\\_campaign=video\\_row1](http://www.infoq.com/cn/presentations/the-culture-and-management-of-western-engineers?utm_source=infoq&utm_medium=videos_homepage&utm_campaign=video_row1)。







原因就是团队文化。我认为，团队文化可以刻意塑造。一个团队的文化改变很难，管理者对团队文化的影响力巨大，他的所有言行都会影响团队文化，因此改变团队文化最简单的办法就是换管理者，但是这在很多场景下显然是不现实的。

## 10.2 组织结构

### 10.2.1 团队规模导致的问题

随着团队规模的扩大，我们通常能见到以下几种严重的问题。

(1) 缺乏信任。由于人数众多，难于管理，只能通过制度、流程、规范、绩效约束。因为一切都是以 KPI 为前提的，那些跟 KPI 无关的工作是很难推动的。公司的执行力确实很强，员工绝对服从，不容易出错，但是效率却非常低下。

(2) 没有责任感。高层管理者忙着开各种决策会议。除了管理者，没有人愿意去决策，因为决策失误将是非常大的责任，而领导决策后，就算出现什么问题，也有领导扛着。结果就会出现大量的架构文档，大量的内部汇报材料，取悦领导的优先级高于服务客户的优先级。如果没有人能够承担责任，那么就靠严格的流程，每个人都严守关口，否则出了问题自己要“背锅”。

(3) 部门墙。跨部门协调还不如与第三方合作。我曾经工作过的一家互联网公司便有这类情况，有的部门宁愿去外部买一个系统，也不用兄弟部门的产品，原因是内部跨部门沟通效率太低。

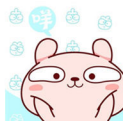
(4) 不尊重专业人士。当所有的生杀大权都掌握在少数人手中的时候，就非常容易形成“一言堂”，领导的话会被仔细推敲，而某些领域的专家可能因为在汇报中，思维不够敏捷，不善于表达，而得不到尊重。

(5) 管理层级太深。管理层级太深导致的问题很多，因为管理者不知道下面的人在干什么，不知道钱花得值不值，只能通过汇报，那就会导致了很多会议和汇报文档。

### 10.2.2 康威定律

“康威定律”（Conway's Law）包括如下四条。

- 第一定律：Communication dictates design，即组织沟通方式会通过系统设计呈现。
- 第二定律：There is never enough time to do something right, but there is always enough time to do it over，即时间再多，一件事情也不可能做得完美，但总有时间做完一件事情。
- 第三定律：There is a homomorphism from the linear graph of a system to the linear





graph of its design organization, 即线型系统和线型组织架构间有潜在的异质同态特性。

- 第四定律: The structures of large systems tend to disintegrate during development, qualitatively more so than with small systems, 即大的系统组织总是比小系统更倾向于分解。

康威定律中最经典的一句话 “Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations.” 即设计系统的组织, 其产生的设计和架构等价于组织间的沟通结构。通俗来讲, 就是什么样的团队结构, 就会设计出什么样的系统架构。如果将团队拆分为前端、后端、平台、DB, 那么系统也会按照前端、后端、平台、DB 结构隔离, 如图 10-1 所示。

如果将团队按照业务领域进行拆分, 系统就会按照业务领域隔离, 如图 10-2 所示。

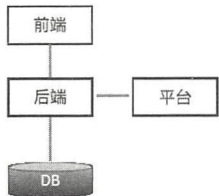


图 10-1 系统架构 (一)

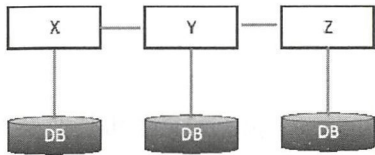


图 10-2 系统架构 (二)

因此当我们在进行架构设计的同时, 也应该相应地调整组织架构。特别是在采用微服务架构的同时, 应该建立全功能团队, 开发、测试、产品应该在一个团队内。项目管理、团队负责人并不一定是专职的, 可以由产品或者开发兼任, 但最好不是一个纯管理人员。某些公司还设置轮岗, 效果不错。

### 10.2.3 扁平化的组织

“沟通漏斗”是指工作中团队沟通效率下降的一种现象: 如果一个人心里想的是 100% 的东西, 当你在众人面前、在开会的场合用语言表达心里 100% 的东西时, 这些东西已经漏掉 20%, 你说出来的只剩下 80%。而当这 80% 的东西进入别人的耳朵时, 由于文化水平、知识背景等关系, 只存活了 60%。实际上, 真正被别人理解了、消化了的东西大概只有 40%。等到这些人遵照领悟的 40% 具体行动时, 已经只剩 20% 了。三个月后信息衰减得有可能只剩下 5% 了。<sup>①</sup>

沟通漏斗模型, 如图 10-3 所示。当管理层级太多时, 就会造成信息传递衰减, 轻则执

<sup>①</sup> 引自智库百科 <http://wiki.mbalib.com/wiki/%E6%B2%9F%E9%80%9A%E6%BC%8F%E6%96%97>。





行力差，重则军心涣散。据说 Amazon 实施微服务架构的初衷就是为了减少沟通，而很多伟大的互联网公司都试图通过扁平化的组织文化来弥补因为公司规模导致的沟通不畅、责任感下降等问题。

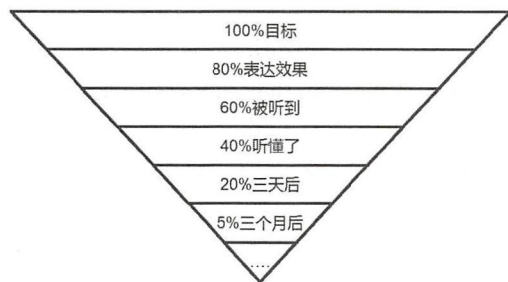


图 10-3 沟通漏斗模型

软件研发型公司应该尽量减少中层管理者，特别是纯粹的“中层管理者”。公司的董事长应该关心中层管理者每天的工作内容，大多数层级比较多的公司，中层管理者工作就是每天不停地开会、汇报。如果董事长要看 1 页 PPT，总裁可能要让下面的总经理准备 10 页 PPT，总经理会让下面的部门经理准备 100 页 PPT，部门经理会让团队负责人准备 1 000 页 PPT。假设，每个 PPT 新建加修改需要 2 天，一年汇报 3~5 次，PPT 产量就会很高。减少一个层级，工作量就会降低十分之一。另外，在这个信息传递的过程中，部门经理为了弄明白小团队负责人的想法，不知道要开多少次会。

## 10.2.4 独裁的管理方式还是民主的管理方式

实际上这个问题并不好回答，因为无论是民主的管理方式还是独裁的管理方式，在当今的商业社会都有成功的案例，也都有失败的案例。

独裁的管理方式更强调个人能力，一位远见卓识的领导者可以将这种管理方式发挥得淋漓尽致，乔布斯就属于这种人。据苹果的员工描述，乔布斯不会给任何员工面子，经常在公司内大发雷霆，在乔布斯面前，没有任何民主可言。下面通过两个故事来描述一下苹果公司的管理风格。

据说新员工入职的时候会安排他们见乔布斯，可以见到自己的偶像员工们都非常兴奋，但是新员工们的热情提问，往往换来的只是冷冰冰的几个字，既简短又粗暴。乔布斯如果觉得问题不好或者不想回答时，干脆直接说下一个。有一位新员工曾问乔布斯：“您觉得最快乐的事情是什么？”乔布斯不耐烦地回了一句：“没有比这个问题更傻的了。”这让提问者无比尴尬。







另外一个故事，据说在 1997 年乔布斯刚回到苹果的时候，因为大规模的“砍项目”，乔布斯把暴君形象发挥得淋漓尽致。那个时候，说不定哪天，某个项目组就会突然被解散，平时在一起办公的同事会突然走过来向你告别。苹果内部流传着一个听来让人不寒而栗的故事，说不定哪天坐电梯的时候就会遇上乔布斯，他会问你：“你叫什么名字？在哪个项目工作？你的工作重点是什么？对公司有什么价值？未来有什么计划？”几乎没有人可以在让人窒息的电梯间里，在乔布斯威严气场的笼罩下，顺利回答上面这几个问题。而一旦员工的回答让乔布斯不满意，员工在电梯里听到的最后一句话就一定是：“好吧，你明天不用来上班了。”

这种独裁的管理方式强依赖于个人能力，当这个企业的领袖能力较强时，这种管理风格高效、快速，拥有强大的执行力，但是如果领袖犯错，那么企业也会跟着走向万丈深渊。因此，这种方式不是每个企业都适合的。虽然乔布斯的管理风格让很多人不舒服，但是乔布斯的能力毋庸置疑，可以说是当时世界上最好的产品经理和市场经理，通过一个人带领一个公司的产品走向成功。乔布斯的产品理念更多的是引领用户，解决了甚至连客户都未曾考虑到的问题，而不是根据用户的反馈来进化产品，这跟很多互联网公司的理念是不一致的。

对于一个独裁者来说，首先，可能自己没有意识到自己正在犯错；其次，用自己过去的经验做决策。第一代 iPhone 受到了当时微软的 CEO 鲍尔默的嘲笑，因为他认为六七百美元的价格太高了，而苹果开创了另外一种营销模式——和运营商合作，按月支付。摩尔定律<sup>①</sup>已经提出了半个世纪，并且似乎还在按照这个趋势发展，谁能保证这个定律一直有效呢？也许未来会打破这个定律。

而民主的管理方式认为世界上不存在完美的人，一个人的决策不可能都是对的，要充分发挥每个员工的能力。由于独裁者通常都比较严苛，下属生怕说错话遭到鄙视、摧残或粗暴对待。因此，这种管理方式无法发挥下属的潜力。民主的管理者更像是一名园丁，通过不断“修剪、浇水、施肥”，让每个人充分发挥出自己全部的天赋，服务大于管理。作为一线员工，你无须关注听命于谁，而是更多的要拥有主人翁精神，在这种民主的企业里，一线员工最重要，这里讲一个 Facebook 的故事。在 Facebook，当几个程序员在一起解决问题的时候，主管会去几公里以外的地方买回来丰富的点心，管理者通常更像后勤人员一样为程序员提供服务。很多企业也会有能力不行的人做管理，显然这种管理者是指挥人员，拥有至高无上的权力。

在民主的企业中，通常创始人或者企业领袖会给公司制定规则，进入公司即代表你认同规则，所有人都要遵守。例如玩一种扑克牌的游戏，假如你感兴趣就一起来玩，但是不

---

① 1956 年，戈登·摩尔提出摩尔定律：当价格不变时，集成电路上可容纳的元器件的数目，每隔 18~24 个月便会增加一倍，性能也将提升一倍。换言之，每一美元所能买到的电脑性能，将每隔 18~24 个月翻一倍以上。这一定律揭示了信息技术进步的速度。





允许任何人违反规则，在规则内可以自由发挥。民主通常伴随着透明、公开，这种文化意味着更强大的创新能力。另外，在民主的制度下，不会去设置更多的障碍和流程，更强调自主决策和责任感，一旦偏离了轨道，就只能被淘汰。

民主也会有很多弊端，例如统一思想的成本会非常高，执行力较差，不容易协作完成一件事。

### 10.2.5 民主的团队如何做决策

前面谈到，民主的团队在决策的时候是痛苦的，即很难统一所有人的意见。如果让该领域的专家决定，那么他的思路有可能不够开阔，没有高度；如果让管理者决定，那么他又可能缺乏该领域的专业知识；如果让所有人投票决定，那么效率又太低。

实际上，我认为一种好的决策机制应该是这样进行的。

(1) 由领域专家做一些简单的前期调研，收集材料，初步计划。

(2) 通过邮件发给决策团队，决策团队成员不应该人数太多。

(3) 开会讨论，大家各自发表意见。

(4) 进行投票，少数服从多数，可以先投出前两名，然后各自发表意见，再针对前两名投票决策。

这是一种非常公平的做法，每个人的意见都得到了充分讨论，如果不能按照自己的意见执行，也应该心服口服。关于这个问题，《原则》一书给了我们很好的启示。

可信度加权决策法：充分考虑到每个人的专业背景，然后对不同专长的人提出的意见赋予不同的权重，最后加权计算进行决策。也就是在针对一个重大决策征询意见时，该领域的专业人士的意见权重就会更大，非专业人士的意见权重更小，比如讨论的是一个医疗问题，有医疗背景的人提出的意见所占的权重就应该更高。

相对于一个人做决策，这种流程还是比较复杂的，需要开会、投票，只有涉及重大问题才有必要采用，否则效率会很低。

## 10.3 环境氛围

### 10.3.1 公开透明的工作环境

如果你撒了谎，就要不断提醒自己曾经和谁撒过谎，这会消耗你相当大的一部分精力，而真实、透明可以让你轻松很多。在企业中也一样，越是公开透明，效率就会越高，就越

能得到理解，满意度就会越高。不要告诉员工，管理者都是天才，是不会犯错的。实际上，公开透明的的工作环境是彼此信任的基础。

每个人每天的工作、会议，包括领导的，公开透明地展示给其他人，当然这不需要通过写工作日志来展示，可以利用工具。如果你是一个技术经理，可以在 Outlook 中共享每天的会议日程；如果你想约见领导，绝对不需要跟领导的秘书沟通，也不用问领导什么时间有空，而是到 Outlook 上查一下领导什么时间有空，如果会议时间和某个人有冲突，完全可以拒绝或者建议修改会议时间。

当然，为了提升工作效率的一切行为都是值得提倡的，工作中，我们常常会被乱七八糟的事情打断，因此，我建议每个员工都应该有不打卡的权利，有不在工位工作的权利。例如，当某个项目紧张的时候，马上面临交付时间点，开发人员为了不被其他人打扰，选择在家办公是应该被鼓励的，你认为这么紧张的时刻他会钻空子吗？当生产环境出了问题需要解决的时候，你认为解决问题的人会去浪费时间吗？如果真的有，那这个人就不应该存在于这个团体里。

### 10.3.2 学习型组织

如果仔细研究企业的生命周期，不难发现，一个企业是很难长时间生存的，“百年老店”屈指可数，无论你现在看到的企业有多么强大，它最终都会退出历史舞台，并且这个时间不会太长。如何才能不断地刺激团队，让团队焕发活力呢？那就需要建立学习型组织，使团队不断改进，获得持续的竞争优势。

并不是只有看书才叫学习，学习充斥在各个角落，例如，Code Review 就是一个相互学习的过程。通过如下方式可以建立一个学习型组织。

- 让团队拥有共同的愿景、目标。NBA 最经典的一句话是“永远不要低估一颗总冠军的心。”<sup>①</sup> 当一个团队有了共同的愿景之后，成员就会忘记自己的角色，只要是对团队有利的就去执行。
- 让团队持续学习。学习不是一次性的，是一个持续的过程。当害怕自己说的话可能是错误的时候，就应该去学习。
- 团队成员平等、自由。只有拥有平等的氛围，才能换位思考，不以过去成功的经验轻易否定一些创新性的想法。

---

<sup>①</sup> 这句话来自时任休斯顿火箭队主教练的鲁迪·汤姆贾诺维奇。当年，他以第六名的身份进入季后赛，在不被看好的情况下，一路杀到总决赛，并获得总冠军。赛后鲁迪·汤姆贾诺维奇说出了 NBA 历史上最经典的一句话“永远不要低估一颗总冠军的心”。



- 团队成员自我驱动、自我管理。不依靠严格的制度管理团队，团队成员具备自我管理的意识。
- 团队积极、有凝聚力。团队成员积极乐观，具有主动性，不推卸责任，团队气氛融洽。
- 团队内部，应该通过不断交换意见获得成长，寻找机会刺激团队成员去思考、分享。
- 团队外部，应该通过一些外部会议、论文、书籍等来开阔团队成员的眼界。

### 10.3.3 减少正式的汇报

试想，如果你在一个大公司，让你实现一个邮件服务，你可能会去学习现有的邮件系统，分析竞争对手，引进竞争对手的人，借鉴竞争对手的架构，使用一堆架构模式、方法论、框架，分析我们如果完成一个这样的系统会在哪些方面超越竞争对手，这是一个大公司的正常逻辑。等我们忙活了半年真正实现了，结果用户可能不会买账，因为没有任何新意，但是我们每次都汇报得不亦乐乎。这就是一个大公司花高额的年薪聘请了高学历的人才做着平庸的产品的生命历程。大家只关心 KPI，而 KPI 是由你的主管决定的，如果按照这个套路至少不会“犯错”，因为所有人都“错”不起，结果顶尖人才不是在汇报就是在写 PPT。

与此形成鲜明对比的是，那些伟大公司的领袖都有减少汇报的要求。乔布斯不允许使用 PPT 汇报，不仅仅是因为做 PPT 浪费时间，还因为乔布斯希望他的团队能够进行激烈的争辩和批判式思考，而不要依赖 PPT。据说比尔·盖茨也不会让下属去给他汇报，他会走到下属的工位，随便聊几句，用一个便利贴记住谈话的结论。这种随机聊天反倒会得到最有效的、没有经过修饰的信息。

有人会问，领导怎么了解进度啊？怎么识别风险啊？怎么判断谁干得好，谁干得不好啊？出了问题该由谁来负责啊？我认为，公司内部的项目管理工具上应该能直观地看到进度，这是动态的，绝对比汇报的内容要更真实、更快捷。软件做得好不好应该由用户去评价，一方面通过用户的反馈意见，另一方面通过自己收集到的数据综合判断，而不是通过几页 PPT 去了解。至于谁干得好，谁干得不好，应该由和他朝夕相处的人去判断。另外，团队成功才是真正的成功，如果害怕大锅饭导致效率低，完全可以换一个更好的搭档。

### 10.3.4 高效的会议

开会的目的和意义是什么？晨会、周例会、月例会、计划会议、总结会议、行政管理会议等各种会议，作为会议的主持者或者团队领袖，在会议中应该扮演什么样的角色？

会议的效率低下，主要可以总结为如下几个原因。

- 参会人数太多。如果一共 20 人参会，平均每人花费 6 分钟，那么两个小时就过

去了。

- 不明白为什么要开会。实际上好多会议的目的并不明确，导致参会的人思维特别发散，会后没有任何成果。
- 会议没有主持人或者引导者，思维太发散。
- 会议中存在分歧，短时间内很难达成共识。
- 会议中的不平等，会议中有的人不敢说话或者插不上话。

会议最主要的目的是达成共识。少数人服从多数人，说服决策者。

**缩小会议范围。**除了培训之外，如果参会的人只是听，而不发言，那么此人不应该参会。不发言意味着他对会议议程和结果不会产生影响，同时他有很多其他途径可以获得会议内容及结论。要么带着问题参会，要么带着信息和观点参会。如果你让他参会就应该给他发表意见的机会，而不是让会议变成“一言堂”，或者是几个思维敏捷者的会议。在这种情况下，会议主持者应该学会自己聆听，分析所有人的意见。一个高效的会议，聪明的会议主持者是必不可少的。他可以让所有参会的人都发挥出最大价值，而不是所有人的观点都偏向某位领导或者意见领袖。此时，整个会议的价值大大降低，只剩下少数人的价值得以发挥。

**常规会议不应该超过 45 分钟。**怎么做到呢？答案就是会前要准备充分。据说在 Amazon 会议前 10 分钟是用来查看文档的，让每个参会的人都了解今天的会议内容，当然这 10 分钟也可以放到会议之外，看完文档直接讨论，这样一般会议时间可以控制在半小时以内。另外，如果存在巨大的分歧，无法形成妥协，最好暂时停止会议，经过一段时间的思考，重新决策，而不是一直在会议中争论。

**限制“意见领袖”的发言时长。**会议主持者要控制“意见领袖”的单次发言时长、发言的频率，以平衡所有参会者的发言时间。关于这一点，可以参考法庭，无论是原告还是被告，都认为自己是正确的，需要轮流发言，如果不加以限制，就会乱成一锅粥。如果存在一个经验丰富的会议主持者，可以不断总结、概括，将会使会议的效率大幅提升。尝试让不爱发言的人先发言，这样有利于不爱发言的人提早进入角色。

**提供平等的会议氛围。**统计表明，机长驾驶飞机出事故的概率高于副机长驾驶飞机出事故的概率，因为机长驾驶的时候，就算副机长发现有什么不对，一般也不敢指出。而副机长驾驶飞机时，机长在旁边监督，一旦发现问题，机长是没什么顾虑的，可以直言不讳<sup>①</sup>。机长就像是领导，往往会议中机长说了太多的话，出问题的概率更高，因此领导更应该学会聆听。

会议中不允许开小差。开会时，如果一会儿接电话，一会儿发邮件，每个人带笔记本

---

<sup>①</sup> 来自《异类》。

电脑不停地打字，或者在底下不停地摆弄手机，那么这个会议将变得非常低效。这时候会议主持者就需要起作用了，时不时地发问，让每个人进入思考模式。总之，开会时所有人都应该聚精会神。

会议中的分歧不应该延伸到会议之外。讨论过程中可以自由发挥，分别阐述自己的思路，会后要保持一致，不应该因为会议中自己的观点没有得到支持而怀恨在心，应该从别人的角度尝试考虑一下。

以上会议规则应该贴在会议室中，时不时地提醒开会的人遵守，甚至要通过预定会议室的时间来限制会议时长。我从未见过超过一个小时的会议还能让所有参会者保持很高的专注度的。总之，会议室是一个企业文化的体现，企业文化的形成需要企业领袖不断地推进、示范。

关于如何高效率开会，可以参考《罗伯特议事规则》<sup>①</sup>，这本书是美国国会开会的规则，应用十分广泛。

### 10.3.5 量化指标致死

量化指标是指能用具体数据来体现的指标，以数据反映人类各种活动。<sup>②</sup>常见的量化指标如 KPI、某个产品的性能等。我并不反对量化指标，但是反对量化指标的考核方式，反对不成熟的量化指标，反对一厢情愿的量化指标。

海底捞的掌门人张勇用自己的经历说明他的 KPI 量化指标是怎么被打败的，值得深思。海底捞的服务好不好？有很多吃饭的人都是冲着它的服务来的。但是以前海底捞的 KPI 是量化的，例如杯子里必须要有水，那海底捞就设定了一个指标，客人杯子里的水必须达到一半。后来发现，有的客人吃完饭要走了，说：“我不喝了”，服务员说：“不行，我得给你满上”。更好笑的是手机套，客人表示不需要，服务员就趁你不注意的时候给你的手机套上，我在海底捞亲眼见过类似情景。

这就是量化指标，也可以说是硬性指标。并不是说量化指标不好，而是说以量化指标来考核，特别容易导致考核畸形，会让被考核的人忘了指标原本的目的。当然这些指标是可以有的，可以用来做报表、做统计、做参考。软硬结合也是不错的，不过千万不要高估自己对指标的理解能力。

话说我曾经工作过的一家公司以前是不打卡的，非常自由，这个公司在业内的口碑还

---

① 《罗伯特议事规则》出版于 1876 年，由亨利·马丁·罗伯特编写，作品内容非常详细、包罗万象，有专门讲主持会议的主席的规则，有针对会议秘书的规则，当然还有大量有关普通与会者的规则，有针对不同意见的提出和表达的规则，有关辩论的规则，还有非常重要的、不同情况下的表决规则。

② 来自百度百科。







不错，实际上大多数人并不会因为不打卡就觉得工作少，打卡与工作量根本没有关系，这个谁都知道。有一天老板上午 11 点到公司发现好多员工刚刚进门，一怒之下，要求员工不但打卡，而且加薪的时候日平均打卡时间必须满 9 个小时。我觉得这样的结果是老板比较安心了，看着指标开心了，而公司的口碑下降了，招人难了，还会有很多员工比较难受，琢磨怎么应付。

再说 KPI 量化的问题。实际上，量化对小团队的管理者是有帮助的，年底绩效考核的时候会有依据，比较容易区分。但是还是有问题，例如，你平均写的代码行数、发布次数、故障率、可用性等，工程师们一定知道怎么绕过。员工多了，真的不好管理，特别是基于不信任的前提下的管理。

对于程序员这个人群，柔性管理往往起到的效果更好。

举个例子，很多刚入职的女程序员认为她们被歧视了，不好找工作，我并不这么认为。如果面试官是我，我会适当降低标准把女程序员招进来，因为我让她来不完全是写代码的，所谓男女搭配，干活不累。有的时候比你给下属打个 A 起到的效果都好。张勇也举过这类例子，他的一个非常优秀的员工离职了，原因是前台那个女员工明确告诉他不喜欢他。

## 10.4 管理风格

团队是由个体组成的，管理风格往往能够显示出团队文化。

### 10.4.1 下属请假你会拒绝吗

很多公司的制度是：请假一天需要直接主管批准；三天以上需要次级主管批准；十天以上需要部门领导批准。

在一个有几百个人的大公司里，可能部门领导对你的工作没有任何印象，他可能不批准你的假吗？不可能！就算你的直接主管可能不批准吗？也不可能！因为你的请假理由一定会很充分。

那么，为什么还要有这个流程呢？因为要防止有人请假导致工作上有依赖关系的人受到影响。那请假流程也并没有通知这些人啊，最终可能还是通过邮件通知的。

还有可能是防止某些没有团队精神的人偷懒，或者是防止他的直接主管和他一起作弊。如果一个公司有 1 万个人，用这个流程只能约束 100 个人，那么这个流程无疑是失败的。你应该在招聘的时候，或者试用期的时候就把这个人开除。

所以，我曾经工作过的一个公司改变了，所有的请假只要发个邮件或者在 yammer（类似公司内部的微博）上通知相关的人就可以了，无须任何审批流程。取消审批之前有很多





人担心会出现各种问题。但是，实施后不但没有遇到任何困难，反而减少了很多麻烦。

## 10.4.2 为什么你招不到你想要的人

有很多人曾经向我求助，让我推荐合适的技术人选，可惜的是，大多数时候，面试是失败的，原因五花八门，我曾经也长期遭受这个困扰，下面将几个关键点总结如下。

### 如何吸引顶级人才

实际上，很多人把吸引不到顶尖的人才的主要原因归结为给不了足够多的钱，给不了足够大的成就感，但是事实并非如此。这里我想用两个故事来说明。大家都知道乔布斯被斯卡利赶出了苹果公司，但是并不知道斯卡利实际上在来到苹果公司之前是百事可乐公司的 CEO。1983 年，百事可乐公司正如日中天，乔布斯用了一句著名的话挖到了斯卡利，这句话就是“你想靠卖糖水来度过余生，还是想要一个机会来改变世界？”而当年蔡崇信放弃了 580 万元的年薪，拿 500 元的月薪加入了阿里巴巴，也让很多人惊讶，当时蔡崇信的收入，用马云开玩笑的话说就是：“蔡崇信可以买下十几个当时的阿里巴巴”。

如何吸引顶级人才呢？首先，你做的事情应该足够有吸引力。其次，你需要通过某种方式吸引到对方。你以为面试只是你在考察别人？面试的过程不只是面试官考察面试者，面试者也会通过面试考察这家公司，考察今后所在的团队，因为往往面试官是面试者入职后的直接领导，面试官决定面试成败。一定要找团队中最优秀的人去面试，被面试的人可以通过面试官了解到这家公司的技术实力，要让被面试的人感觉到面试官很专业，水平很高，是一路人，这个团队值得加入。此外，招人不能只靠薪资，很多时候薪资并不是决定因素。

### 为什么要么招不到人，要么招到“凑合的”人

很多公司，特别是大公司，招人只看短期，没有长期规划，突然有几个名额，就要立马到岗，因此把大部分招聘时间都花在了流程上。按照这种方式，很难招到理想的人。招聘是一个长期规划，不是缺人了才招，况且看过《人月神话》的人应该都知道，项目中加人不等于加快进度。很多公司越是忙了越招人，还得培养人。我认为科学的做法是，招聘不设定名额，长期招聘，遇到符合条件的人就招。因为这个行业的离职率非常高，超编的情况太少见了。要么你长期缺人，要么你招到的人距离要求很远。当然，你可以在人员充足的时候适当提升一下招聘要求，这是一个良性循环。

### 招聘比你优秀的人

一个才华横溢的队友，对团队的影响力，远远超乎想象。如果找到的只是跟你差不多，





或者比你更差的人，那么团队自身提升的速度也会非常慢。和卓越的人合作时，他们会激励身边的同事，团队会迸发出更强的战斗力，远远超越了人数的累加。

在 Google 首席人才官 Laszlo Bock 所著的《重新定义团队》中，详细描述了他们招聘人才的标准——只聘用比自己优秀的人。Google 是如何做到的呢？他们有两个方法，第一个是长期招聘，宁缺毋滥，在人力资源层面，招聘永远都放在第一位；第二个是只聘用在某些特定方面比你优秀的人。

## 空间、成长

实际上很多面试者最关心的问题并不是薪资，而是成长，如果你现在在国内某家公司工作，让你去 Google 工作两年，不给工资，我相信也能招到很多高手。那是因为他们考虑的是从 Google 出来的时候是什么状态。当然，这并不等同于必须去一家带光环的大公司，小而美的创业公司也能吸引高手。大的公司也有劣势，凡是在大公司待过的核心技术人员都知道，你大多数时间是在汇报，在说服你的领导，真正留给你做研发的时间寥寥无几。特别是层级比较深的公司，你可能要给你的领导汇报，给领导的领导汇报，给领导的领导的领导汇报……所以，这一点是我吸引大公司出来的高手的一个制胜法宝，你专心做研发，其他所有的事我尽最大努力帮你解决，不轻易打扰你。

## 你要找的不是一个完美的人

除非你有最高的薪资、最好的公司、最好的项目、最好的团队，还有最好的运气，否则你要的人不一定能招到，就算你招到了，工作两个月，你发现他也不一定像你想得那么好。招人就像找对象，在你爱他的优点的同时，也要爱他的缺点。决定两人长久相处的往往不是非常欣赏他的某些优点，而是能忍受他的缺点。

## 跳槽频繁的人不一定就不能要

很多面试官对频繁跳槽这点非常介意。我发现一些跳槽频繁的人不一定是对环境挑剔，有的时候是内向，不善于表达，不好意思问面试官太多问题，导致工作的时候才发现这个职位不是他想要的。实际上，面对这类人，你只要把所有的情况主动跟他说明白了，他不一定会那么快就跳槽。比如加班问题，绝大多数面试者都非常介意，但是没有几个面试者好意思问的，如果你面试的时候就说明白了，让对方充分考虑清楚，那么因加班而跳槽的情况可能就不会发生了。







## 不要迷信大公司出来的精英

大公司的背景可以作为参考，但是不能太看重。很多大公司出来的人技术不一定很好，而很多名不见经传的小公司也能培养出能够深入产品、运营、研发、运维各个方面的优秀人才。实际上，大公司也有一堆烂代码，相反一些小公司也有非常优秀的代码。在这个开源的世界，秘密已经越来越少了，不要迷信大公司。很多大公司的架构师已经很多年不写代码了，所谓的能力大多只限于理论，你敢让他带“兵”去“冲锋陷阵”吗？

## 试用期才是真正的考察时间

大多数公司的试用期形同虚设，而面试过程短短几十分钟，很难看穿一个人，却淘汰了很多很多人，但是很少有公司试用期淘汰人。难道面试官真能看得那么准，还是出于情面，这是一个非常大的问题。公司完全可以在准备招聘某人的时候说明，试用期间会考察得比较严格。

下面是亚马逊中国在拉勾网上的招聘信息<sup>①</sup>，这是我个人认为比较好的招聘信息。

我们希望你：

- 代码能人，能用 C++/Java/C#等任意一种面向对象的语言交付高质量的代码；
- 架构师，懂大规模分布式系统架构，并且能完成高质量的面向对象设计；
- 问题终结者，对互联网产品充满热情，善于解决实际技术问题，致力于提高用户体验。

我们提供以下帮助。

- 工作与生活平衡，这里没有朝九晚九一周六天的疯狂加班。我们不拿 IPO 计划和摸不着的股票期权来换取员工无休止的奉献。我们提供现金和市值超过 1 400 亿美元的优质股票——过去五年增长率超过 323%！我们知道除了改变世界，大家还有女朋友、男朋友需要照顾，有家人需要陪伴，有兴趣爱好需要投入时间，而这些最终可以促成高质量的交付。
- 高水平的工程师队伍。亚马逊的招聘百里挑一，和你一起工作的工程师绝对都是技术过硬、讲究团队合作、对产品和专业充满热情的小伙伴和大朋友。我们还提供全球导师系统、定期技术论坛和活跃的内部工程师社区，在这里你有很多机会和全球的技术大牛交流想法。

---

<sup>①</sup> 来自 <https://www.lagou.com/jobs/3467603.html?source=pl&i=pl-4>。





- 懂技术、愿意和你一起写代码的高级开发经理。我们的开发经理不玩政治玩代码，他们既不坐观其成，也不对你的工作指手画脚。他们懂你的痛点、兴奋点，和你一起看系统架构，一起攻克技术难关；他们关心产品也关心你，而这一切，都是在招聘他们的时候就设定好了的评判标准。

我们不要的人：

- 不愿意对自己的代码精益求精，不考虑其他人的维护成本和将来扩展需求的人；
- 对互联网行业本身缺乏热爱，不愿意“费尽心机”地用技术解决方案来讨好消费者的人；
- 拒绝学习新知识，不愿意尝试用最有效、最先进的技术（虽然两者有时并不统一，但都需要投入精力学习）来解决问题的人。

### 10.4.3 得到了所有人的认可，说明你并不是一个好的管理者

如果好人认为你是好人，你做对了。

如果坏人认为你是坏人，你做对了。

如果好人认为你是坏人，你做错了。

如果坏人认为你是好人，你做错了。

作为一个有追求的管理者，你肯定希望自己得到所有人的认可，我想告诉你的是，如果你让所有人都满意了，那你一定不是一个出色的管理者。让所有人都满意意味着你放弃了一部分优秀成员的利益，那么这个团队是没有追求的。

确定团队成员是管理者的责任，如果你做得不好，那么队员可能会想方设法自己选择，甚至是换一个公司工作。

### 10.4.4 尽量避免用自己的权力去做决策

虽然你有权去做决策，当然也有可能你有这个决策的能力，无论成败你都一个人承担，但是如果你这么做了，我认为你不是一个好的管理者，因为一个人的能力是有限的，有可能你有自己非常擅长的领域，但不可能所有的领域你都强于你的下属。这样做的一个后果是，所有的队员在未来需要决策的时候，都失去了思考的原动力，因为他认为自己不是一个决策者，而是一个参谋，参谋的目标通常是和决策者保持一致。

可以说，一个管理者对下属的最大尊重就是聆听，认真听每一句话、每一个观点。如果你没等队员说完，就认为已经理解了对方的意思，只能说明你愚蠢至极。据我观察，在大多数场景下，下属总是能仔细聆听领导的意见，并且详细分析。如果你认可下属的能力，





那么你也应该仔细聆听，否则，这个人不应该成为团队中的一员。

管理做到极致就是发挥下属百分之百的价值，而用权力去做决策的时候，实际上只发挥了自己的价值。

### 10.4.5 一屋不扫也可助你“荡平天下”

作为管理者的你，肯定希望下属都是听话的，对你毕恭毕敬的。你是否能容得下一个特立独行的人？他总是在很多时候跟你唱反调，引得团队成员一阵阵哄堂大笑，他似乎是对你权威的挑衅。你肯定担心长此以往其他听话的人岂不是也要跟着学？但是，一个团队最可怕的不是意见不一，而是出奇的默契。管理者的任何一个观点所有人都举双手赞成，还得举几个正面案例证明一下。在这样的场景下，你一定认为自己特别聪明，一语道破“那几个笨蛋”多少年都没弄明白的事。团队到了这一步，战斗力就只剩你自己了。敌人变成战友，多半是为了生存。

如何对待个性强的员工是最能体现管理智慧的，剔除是最简单的做法，也是最普遍的做法。但是剔除可能会让你失去一个利用他的特长的机会。以前我曾经有一个下属，特别喜欢迟到，一个月基本上有一半的工作日迟到，但是，若论总工作时长，他比任何人都多。而我的另一个下属，工作能力特别强，他几乎从不加班，无论工作量多少，每天都正点下班。还有一个下属，他自身能力特别强，对别人的要求也特别高，他会认为，这么简单的问题还要过来问我，这么简单的工作两天不就解决了？所以他经常和其他人产生矛盾。但是这几个人最后都变成了我的左膀右臂。

如果有人过来问我：“为什么他总是迟到还能得高绩效？”我会说：“如果你的总工时超过他，像他一样每天处理各种问题，我也会允许你迟到。”

如果有人过来问我：“为什么他每天都按时下班还能得高绩效？”我会说：“他现在所负责的模块，如果你能承担，而且效率不比他低，我也会同意你每天按时下班，并且得高绩效。”

如果有人过来问我：“为什么他是团队的‘老鼠屎’，影响团队氛围，你还给他高绩效？”我会说：“如果你一年也能完成和他同样多的代码行数，我也允许你对其他同事趾高气扬。”

据说马化腾也是这样容忍张小龙的，张小龙总是以“早上起不来”为借口不参加例会。为了迁就他，马化腾会早上让秘书把他叫醒，派车到楼下去接他。

世界上没有完美的人，每个人可能都会存在这样、那样的缺陷。如果能用好对方的优点，那么即便是一屋不扫，也足以助你“荡平天下”。

但是，这也不意味着可以无限制的纵容下属，但凡涉及人品、用户的时候，我绝对不会心慈手软。假如总是迟到，却找各种理由推脱，自认为自己劳苦功高，因为迟到导致线







上的问题没有及时解决，绝不能纵容。

#### 10.4.6 如何留下你想要的人

**充分沟通。**大部分的人是比较被动的，如果你在会议室说一句，大家有什么问题随时找我沟通，那么往往没有人主动找你沟通。但是如果你找到某个人一对一沟通，则又会发现实际上他们都有很多想法。沟通也是一种发泄的方式，如果沟通顺利，则有可能解决对方的疑问，最好不要让疑问在他的心理发酵。对于沟通能力比较弱的人，当你安排一个任务的时候，最好在会议结束前，复述一下任务，避免当对方完成任务的时候，才发现双方理解不一致，这是非常打击士气的。

**给予尊重。**尊重就是仔细聆听对方的讲话，不打断对方，站在对方的角度考虑问题；尊重就是对专业的人员在专业领域知识的肯定，业余的人去羞辱一个专业的人是极其不尊重对方的。特别是汇报的过程中，有可能是在紧张的氛围中，对方的思路没有你敏捷，但是这并不代表你比对方优秀。如果连你都不尊重对方，那些对你溜须拍马的人会变本加厉，最终导致团队缺乏对专业知识的敬畏之心。受委屈也是离职的一个重要因素。公司内大家只是分工不同而已，没有尊卑，这是绝大多数人的真实想法。尊重不是表面上的，而是发自内心的，如果你从内心就不愿意尊重对方，要么是你太自大，要么是你没有雇佣到合适的人。

**肯定工作成绩。**既然是你想留下的人，工作结果一般不会太差。无论对方的工作结果有多么糟糕，但是总有闪光的地方，总有值得肯定的地方。更多的通过正向激励而不是惩罚的方式来和下属交流。下属完成了很多工作，最好你要一一列举，否则他会认为你并不知道，觉得自己白干了。

**设定合适的岗位。**要绝对避免把不合适的人放到不喜欢的岗位。一个高级开发的岗位，如果你安排了一个实习生，则会不断地打击他的自信心，让他觉得胜任不了工作，最终选择离开；如果你安排了一个架构师，则会让他觉得工作没有任何挑战，最终会去寻找更有挑战的工作。最好在员工入职的时候，就充分沟通未来的发展方向。如果人家想往技术方向发展，就不要安排太多管理工作；如果人家想往管理方向发展，就不要安排太多特别有挑战性的技术问题。

**设定更大的挑战。**很多有能力的人离开都是因为舞台太小，不能充分展示自己。当你拥有一个能力较强的下属的时候，可能你会成为他的发展瓶颈，此时应该给他更有挑战性的工作。例如交给他某个技术难题去解决，或让他代表团队给更大的领导汇报工作。





## 10.5 经典案例

### 10.5.1 Instagram 的团队文化

首先简单介绍一下 Instagram，它是全世界最大的图片分享社区，目前有 5 亿多的日活用户和 8 亿多的月活用户。2012 年，它被 Facebook 以 10 亿美元的价高收购，这场收购当时被认为存在巨大泡沫，但是到了 2017 年的时候，Instagram 估值达到了 350 亿美元。

Instagram 创始人亲力亲为，讲究快速发布产品，每个领域都有独当一面的专家，包括视频、存储、Android、iOS 等，这里的专家是要从需求到上线独立负责的，而不是指挥的。

Instagram 秉承的团队文化如下。

- 化繁为简。Instagram 一直以来的原则都是，先做简单的事。每一次面对新的挑战的时候都需要确定一个最快速、最简单的方法，选择最重要的问题去解决，选择最简单的方法，这样才能快速有效地解决指数级增长的流量所带来的问题。
- 社区至上。Instagram 产品的核心就是社区，一切围绕产品的定位进行。
- 注重细节。每一个功能都做到极致，特别注重用户体验方面。需要说明的是，只有化繁为简，才能更注重细节，否则根本没有那么多精力。

这实际上和微信的团队文化类似，微信第一个版本发布的时候，App Store 评论区里都是不屑的声音，而现在，App Store 评论区里全都是羡慕的声音。微信的功能并不是特别复杂，只是非常注重用户体验罢了。

### 10.5.2 Netflix 的团队文化

NetfliX 成立于 1997 年，是一家在线影片租赁提供商，总部位于美国加利福尼亚州洛斯盖图。Netflix 非常善于利用大数据，曾经根据用户数据制作出热播剧《纸牌屋》。2009 年，Netflix 发布了一个多达 100 页的 PPT，讲述了 Netflix 的企业文化，这份 PPT 被 Facebook 公司的 COO 桑德伯格称为“硅谷最重要的文件”。同时，Netflix 的企业文化，被很多国内一线互联网公司研究借鉴。

那么，Netflix 的企业文化有什么独特之处呢？下面几点是 Netflix 一直坚持的。<sup>①</sup>

#### 鼓励员工自主决策

这与我们前面讲到的独立自主是一致的。Netflix 希望员工都能成为出色的独立决策者，只有在不确定决策是否正确时才咨询管理者。注意，这里是咨询，而不是汇报。Netflix 并

---

<sup>①</sup> 以下内容参考 Netflix 官方博客 <https://jobs.netflix.com/culture>。



不提倡那种 CEO 或者其他高位领导者过度参与产品或者服务的细节工作。Netflix 不是自上而下的管理模式，因为 Netflix 相信只有员工能够自行决策，才会成就最具效率及创新能力的公司。Netflix 支持员工向管理者明确表达“我知道你不同意，但我还是要这么做，因为我认为这种解决方式更好。如果你能够具体地说出我的决定中存在的问题，请告诉我”。Netflix 不希望员工先猜测其管理者会做什么或者想要什么，再去按那样决策执行。在这里，管理者更像一位足球场上的教练，确保发挥出每个位置最大的能力，并且高效配合，而不是让教练告诉每位球员应该怎么踢。

### 开放、广泛、积极地进行信息共享

在 Netflix 内部，几乎所有文件都是完全公开的，每位员工都可以阅读和评论，同时确保评论可供他人查看。每一项绩效指标、每一轮战略决策、每一位竞争对手甚至每一项产品功能测试的备忘录都向员工完全开放。虽然员工众多，但是 Netflix 的员工具有自律和责任感，极少会出现信息泄露问题。在 Netflix 内部，没有任何必须依赖的监管部门。无论在任何团队，合作与信任都是很重要的，因为每个人不仅要擅长处理自己的份内任务，同时还需要与其他同事进行协作。“愿意抽出时间帮助同事”是 Netflix 文化中非常好的一面。

### 非常坦诚地对待彼此

Netflix 文化非常强调坦诚、真实、透明，以及非政治性等特质，坦诚的效率一向是最高的，对待所有人都可以直言不讳。犯了错误，能够自由而公开地承认错误，这一点往往是从上级开始会比较好。尊重他人，且不受其具体地位或者所持观点的影响。

### 只保留高效能人员

在互联网公司，永远不可能把手中所有的工作都做完，因为这个行业发展太快。互联网公司的大多数产品都需要迭代前进，无法定义一个静态的完成时间。一个高效能人员懂得如何提升效率，善于寻找最重要的工作，并且优先完成。摒弃伪工作者，即那些虽然看起来很忙，但是没有工作结果的人。

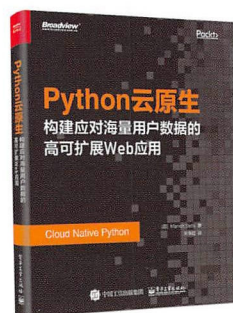
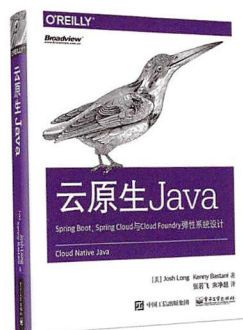
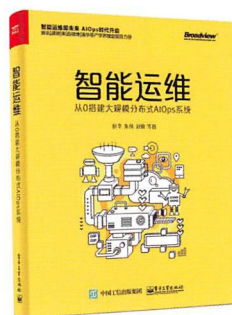
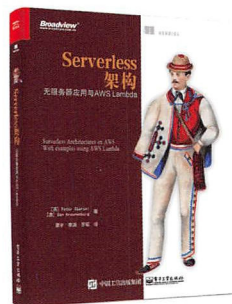
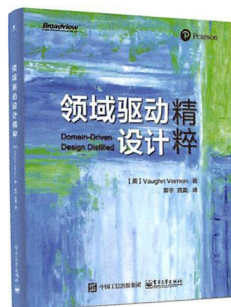
### 避免规则束缚

Netflix 是一个互联网公司，强调创造力而不是稳定性。互联网公司如果失去了创造力会很快被取代。而流程对创造力的破坏是最大的，因此在 Netflix 的文化中，一直强调自由和责任，人的重要性大于流程。例如某些公司的末位淘汰制，对团队的损害实际上非常大，是不可取的。





/ 好书分享 /



## 好评袭来

本书总结了作者的工作经验，望读者在借鉴技术细节的同时，也不忘作者的初心：微服务架构是持续演进的，更是一种开发文化的体现。

——徐振中  
Netflix架构师

以往的云原生书籍尚未形成对云原生理论的系统介绍，也缺少云原生的最佳实践，而本书分别从架构、研发流程、团队文化三个角度论述云原生，为企业或组织从单体架构过渡到云原生架构提供了最佳实践，具有现实的指导意义。

——宋净超  
Cloud Native Go的译者、蚂蚁金服架构师

启军在书中提到“好的架构是锤炼出来的，而不仅仅是设计出来的。”对这句话我深信不疑，也一直这样践行。锤炼一个好的架构需要掌握非常丰富的知识，一本好书可以加快我们的成长速度。启军是我非常敬佩的架构专家，他能把Cloud Native这个非常大的概念讲得明白、透彻，可见其功力之深厚。我坚信本书一定是读者们期盼已久的。

——黄勇  
《架构探险》作者、特赞科技CTO

在实施微服务时，最容易犯的错误就是，只关注微服务技术框架本身，而丢失了全局观。如果没有配套的微服务流水线、基础设施自动化，以及对应的服务化团队和组织结构，那么微服务很难达到预期收益。本书从全局视角出发，内容涵盖了Cloud Native的关键技术，以及其衍生出的工具、团队文化等关键要素，对于开展云原生业务的企业和组织具有极高的参考和学习价值。

——李林峰  
《分布式服务框架原理与实践》作者、  
华为消费者云微服务架构师

从单体架构演进到云原生架构的技术变革，是为了解决快速变化的业务带来的不确定性问题。简而言之，有什么样的业务就有什么样的技术与其相适应。但是仅仅知道这样一个简单的结论，并不能指导我们解决工作中遇到的实际问题。本书比较全面地介绍了微服务架构演进的原则与实践，提供了分布式架构带来的数据一致性问题解决方案。希望本书能帮助读者解决不同的业务场景下遇到的实际问题。

——万金  
《DevOps实施手册：在多级IT企业中使用DevOps》译者，曾在IBM、华为和ThoughtWorks从事DevOps的实施与推广工作

启军有着丰富的软件开发和管理经验，对软件开发架构设计有自己独到的见解，书中的内容都是他从十几年的工作中获得的实战经验。这绝对是一本值得认真阅读的好书，我强烈推荐！

——马宝辉  
亚马逊中国数据工程师

Pivotal公司于2013年提出云原生（Cloud Native）之后，先后开源了云原生的Java开发框架Spring Boot和Spring Cloud，随后Google在2015年成立了CNCF，使云原生受到越来越广泛的关注。云原生不仅仅是研发技能或架构设计，它的本质是一种模式，它要求云原生应用满足可用性和伸缩性，具备自动化部署和管理能力，可随处运行，并且能够通过持续集成及持续交付工具提高研发、测试与发布的效率。本书细致地将云原生的理念、所需技术、研发流程及团队文化融合在一起，能够帮助读者快速准确地理解云原生的精髓。

——张亮  
京东金融数据研发负责人、开源分布式数据库中间件  
Sharding-Sphere的负责人



博文视点Broadview



@博文视点Broadview



策划编辑：张春雨  
责任编辑：汪达文  
封面设计：李玲

上架建议：计算机 / 编程语言

ISBN 978-7-121-35120-4



9 787121 351204 >

定价：79.00元